# The Selection and Single Event Upset Testing of a DSP Processor for a LEO Satellite

Heiko Berner

UNIVERSITEIT VAN STELLENBOSCH
UNIVERSITY OF STELLENBOSCH

*Thesis presented in partial fulfillment of the requirements for the degree of Master of Engineering at the University of Stellenbosch*

Supervisor : Dr Mike Blanckenberg

March 2002

# Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

# Abstract

After successful use of a DSP processor onboard the SUNSAT satellite, the need arose for a faster floating-point processor. A list of possible processors was generated from various selection criteria. Two suitable DSP processors were chosen, and because no radiation information was available for one of them, the decision was made to perform radiation tests on it. The procedures used to test the processor are described in detail so the same methods can be used for future radiation tests. An error detection and correction circuit was implemented to check and correct upsets in the on-chip memory of the DSP processor. This ensures that the processor code and data stays intact.

# Opsomming

Na suksesvolle gebruik van 'n DSP verwerker aanboord die SUNSAT satelliet het die behoefte ontstaan vir 'n vinniger wissel-punt verwerker. 'n Lys van moontlike verwerkers is opgestel met die hulp van verskeie seleksie kriteria. Twee geskikte DSP verwerkers is gekies, en omdat geen radiasie informasie vir die een beskikbaar was nie, is besluit om radiasie toetse op hom te doen. Die prosedures gebruik om die verwerker te toets word deeglik beskryf sodat dieselfde metodes in die toekom gebruik kan word. 'n Fout deteksie en korreksie baan is geimplementeer om foute in die aanboord geheue van die DSP verwerker op te spoor en te korrigeer. Dit verseker dat die verwerker se kode en data intak bly.

# Acknowledgements

I would like to thank Dr Mike Blanckenberg for the support he gave me throughout the thesis. Then all the people at the NAC who made the radiation testing possible and helped with the tests. They are Dr Kobus Lawrie, Mr E.A. de Kock, Mr J.E. Symons and Dr D. Nichiporov. Thank you very much to my mother and father who held out with my writing in the house and the support they gave. All my friends in the lab, thanks for being there when I needed help. Lastly thank you Lord for making this all possible.

# Contents

# List of Figures

# List of Tables

# List of Acronyms and Abbreviations

| | |
|---|---|
| A | Ampere |
| A/D | Analogue to Digital |
| ADCS | Attitude Determination And Control Systems |
| ALU | Arithmetic and Logic Unit |
| ASM | Assembler |
| BDTI | Berkeley Design Technology Incorporated |
| cm | Centimetre |
| Coax | Coaxial |
| COTS | Commercial Off The Shelf |
| CMOS | Complementary Metal-Oxide-Semiconductor |
| D/A | Digital to Analogue |
| dB | Decibel |
| DC | Direct Current |
| DM | Data Memory |
| DSP | Digital Signal Processing |
| EDAC | Error Detection And Correction |
| EPROM | Erasable Programmable Read Only Memory |
| ESC | Escape key |
| eV | Electron Volt |
| FET | Field Effect Transistor |
| FIR | Finite Impulse Response |
| FPGA | Field Programmable Gate Array |
| GeV | Giga Electron Volt |
| GHz | GigaHertz |
| GPP | General Purpose Processor |
| I/O | Input/Output |

| | |
|---|---|
| JTAG | Joint Test Action Group |
| K | Kelvin |
| kbit | kilobit |
| keV | Kilo Electron Volt |
| LET | Linear Energy Transfer |
| LSB | Least Significant Bit |
| MAC | Multiply-ACcumulate |
| Mbit | Megabit |
| MeV | Mega Electron Volt |
| MFLOPS | Million FLoating-point Operations Per Second |
| MHz | Megahertz |
| MIPS | Million Instructions Per Second |
| mm | Millimetre |
| MOS | Metal Oxide Semiconductor |
| MOSFET | Metal Oxide Semiconductor Field Effect Transistor |
| MV | MegaVolt |
| NAC | National Accelerator Centre |
| NOP | No OPeration |
| NRF | National Research Foundation |
| PC | Personal Computer |
| PM | Program Memory |
| Pot | Potentiometer |
| RAM | Random Access Memory |
| RF | Radio Frequency |
| ROM | Read Only Memory |
| SRAM | Static RAM |
| SEGR | Single Event Gate Rupture |
| SEL | Single Event Latchup |
| SEU | Single Event Upset |
| SHARC | Super Harvard ARchitecture Computer |
| SIMOX | Separation by IMplanted OXygen |
| SSC | Separated Sector Cyclotron |
| SUNSAT | Stellenbosch UNiversity SATellite |
| TID | Total Ionising Dose |
| V | Volt |

| | |
|---|---|
| VHDL | VHSIC Hardware Design Language |
| VHSIC | Very High-Speed Integrated Circuit |
| XOR | eXclusive OR |

# Chapter 1

# Introduction

In 1999 the first South African satellite called SUNSAT was launched on a Delta II Launch Vehicle. The acronym SUNSAT stands for Stellenbosch UNiversity SATellite. It was the result of a group of postgraduate engineering students doing their masters degrees on the project. The satellite was used for various applications in its lifetime, and one of the components that played an important role on the satellite was a DSP processor. The functions that the DSP processor provided proved to be invaluable. This led to the need to select a DSP processor for a future satellite. The new processor had to be faster than the previous one and be able to do floating-point arithmetic as well. There are many other important factors in the choice of a DSP processor as well, which are discussed in the thesis.

Suitable DSP processors were chosen and because one of them had not been radiation tested, the decision was made to perform such tests on it. Background information on radiation effects and environments are given in the thesis. The preparation involved with the radiation testing comprised the largest part of this thesis. The procedures used to test the processor are described in detail so future radiation tests can use the same methods.

An Error Detection And Correction (EDAC) circuit was developed to ensure that the processor could be used for space applications if the radiation test results were positive. The purpose and functioning of EDAC codes are covered, with detail on the Hamming EDAC code. The implementation of the EDAC circuit is described as well, which provides a method to check the on-chip memory of the DSP processor.

Chapter 2 covers the criteria for selecting a DSP processor and the results after the criteria were applied. An overview of the radiation environment with all the effects it has on semiconductors is given in Chapter 3. The process of doing SEU testing with a description of the hardware and software used is discussed in Chapter 4. The results of the tests are covered as well. Chapter 5 contains the design of an EDAC circuit to check the internal SRAM of the DSP processor. The conclusions are summarised in Chapter 6.

# Chapter 2

# Choosing a DSP Processor

This chapter describes some of unique features of DSP processors and which aspects are important when selecting one. A short list of processors is generated and the selection criteria discussed is used to reduce the list. Two processors from the list are chosen as best suited for use onboard future satellites. The two processors and their evaluation boards are described briefly at the end of the chapter.

## 2.1 Introduction to Digital Signal Processing (DSP)

The definition of digital signal processing (DSP) and its advantages are discussed at the start of the section. The differences between DSP- and general purpose processors are covered thereafter.

### 2.1.1 What is Digital Signal Processing (DSP)

Digital Signal Processing (DSP) refers to the processing of signals by a computer or other digital system. This can be defined informally as the application of mathematical operations to digitally represent and process signals. Usually analogue signals are converted to digital ones by analogue-to-digital converters. After the data has been processed, the data can optionally be converted back to analogue by a digital-to-analogue converter. [23]

### 2.1.2 Advantages of DSP

DSP has several advantages over analogue signal processing and enhances, or even replaces, analogue signal processing. DSP adds more decision-making ability to signal processing. This allows functions like speech recognition and high-speed modems with error-correction coding. These are very difficult, or impossible, to implement with analogue circuits. More advantages of DSP over analogue signal processing are listed below:

- Analogue circuitry is subject to behaviour variation depending on environmental factors such as temperature. Digital circuits are essentially immune to such environmental effects.
- DSP applications are easily duplicated within tight tolerances, since their behaviour do not depend on a combination of components, each of which deviates to some degree from its normal value.
- Once an analogue circuit is built, its defining characteristics (such as a filter's passband) are not easy to change. By using a DSP processor, the characteristics can be changed by simply reprogramming the device. This allows the same device to be used for more than one task.
- The size of analogue circuits depend on the number and size of the components necessary for the application. DSP circuits usually have the same size, no matter what the application. [10] [23]

### 2.1.3 Difference between DSP- and General Purpose Processors

A DSP processor is a programmable processor that is particularly efficient at implementing DSP algorithms. These algorithms are very math-intensive and require high processing speed. General Purpose Processors (GPPs) are used for applications which usually do not require high amounts of mathematical calculations. There are many more distinctions between DSP processors and GPPs. The differences in mathematical abilities, memory architecture, special instructions and execution time predictability are discussed below.

### Special Mathematical Hardware

GPPs were not originally designed for multiplication-intensive tasks. Even some modern GPPs require multiple instruction cycles to complete a multiplication because they lack dedicated hardware for single-cycle multiplication. This was the first major architectural modification that distinguished DSP processors from the early GPPs.

DSP processors have specialised hardware to enabled single-cycle multiplication and accumulator registers to hold the summation of several multiplication products. These modifications make the multiply-accumulate (MAC) operation possible. A MAC operation multiplies two values and accumulates the result with a previous result, usually all in a single instruction cycle. This is used for example to implement a Finite Impulse Response (FIR) filter which may be expressed as:

$$y(n) = \sum_{m=0}^{M} a(m) \cdot x(n-m)$$

FIR filters entail that two operands, $a(m)$ and $x(n-m)$, are multiplied and then accumulated with the previous results $\sum$. This operation is usually carried out in a small loop to compute all the sums. An efficient DSP processor will have a MAC instruction which is executed in a single processor cycle. To operate efficiently, the processor has to provide the multiplier-accumulator with new data to process each cycle. [27] [10]

### Memory architectures

Another difference between DSP processors and GPPs is their memory structure. Traditionally GPPs have used a von Neumann memory architecture. In the von Neumann architecture there is one memory space connected to the processor core by an address bus and a data bus. See Figure 2.1(a) for a diagram of the architecture. The von Neumann architecture works well for most computing applications, since the memory bandwidth is sufficient to supply the processor with instructions and data. Typical DSP algorithms require more memory bandwidth than the von Neumann architecture can provide.

For example, the processor must complete one MAC and make several accesses to memory to sustain a throughput of one FIR filter tap per instruction cycle. Thus, the processor must make a total of four accesses to memory in one instruction cycle. In a von Neumann memory architecture, four memory accesses would consume a minimum of four instruction cycles. For this reason, instead of a von Neumann architecture, most DSP processors use some form of Harvard architecture. [10]

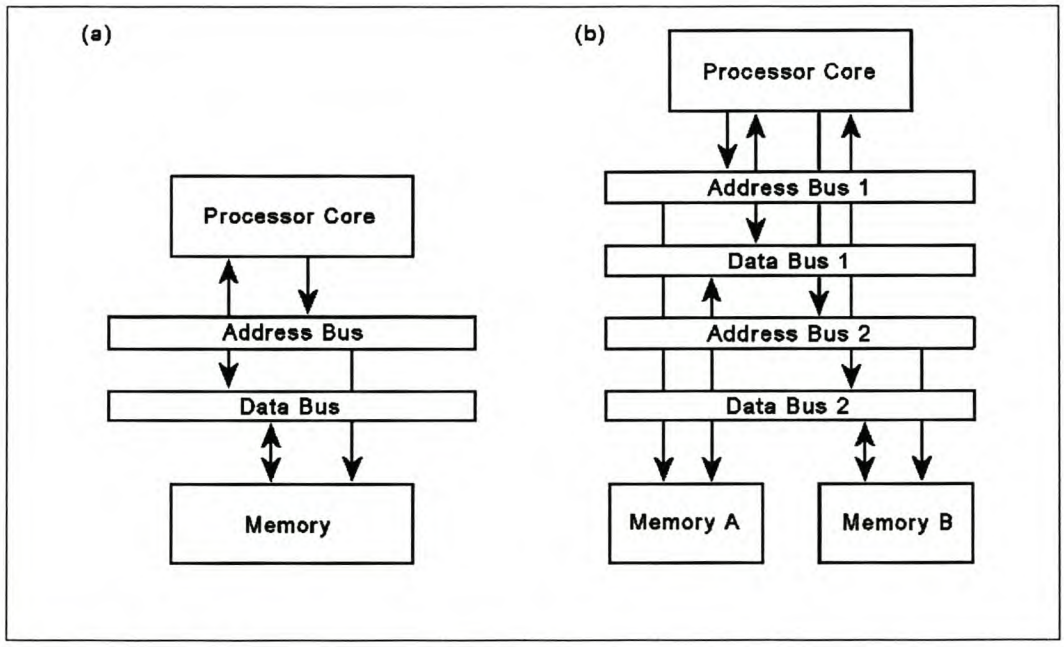



**Figure 2.1:** **(a)** Many general purpose processors use a von Neumann memory architecture which permits only one access to memory at a time. **(b)** DSP processors typically use a Harvard memory architecture which permits multiple simultaneous memory accesses through the two independent sets of buses. [23]

In a Harvard memory architecture there are two memory spaces, typically partitioned as program memory and data memory (though there are modified versions that allow some crossover between the two). The processor core connects to these memory spaces by two bus sets, allowing two simultaneous accesses to memory (See Figure 2.1(b) for a diagram). This arrangement doubles the processor's memory bandwidth, and is crucial to provide the processor with data and instructions. The Harvard architecture can be extended with additional memory spaces and/or bus sets to achieve higher memory bandwidth. The trade-off is that extra bus sets require extra power and chip space, so most DSP processors use only two.

Many high-performance GPPs have clock rates higher than 200 MHz. Memory chips capable of running at these speeds are very expensive, and these GPPs often need to make multiple memory accesses per instruction cycle as well. One way to solve both these problems is to use on-chip cache memory. Typically two on-chip memory caches are used, one for data and one for instructions. When required information resides in the cache, the processor retrieves instruction and data words without accessing slow off-chip memory. Most DSP processors do not have any caches, although some include a small, on-chip instruction cache. This cache stores instructions used in small inner loops so that the processor does not have to use its on-chip bus sets to retrieve instruction words. DSP processors seldom use a data cache, since data samples are usually used once, and then discarded during a computation. [10]

### Zero-overhead looping

One common characteristic of DSP algorithms is that most of the processing time is spent executing instructions contained within relatively small loops. In an FIR filter, for example, the majority of processing takes place within a very small inner loop that multiplies the input samples by their corresponding coefficients and adds the results. This is why most DSP processors include specialised hardware for zero-overhead looping. The term zero-overhead looping means that the processor can execute loops without consuming extra cycles. The hardware tests the value of the loop counter, performs a conditional branch to the top of the loop, and decrements the loop counter. In contrast, most GPPs do not support zero-overhead hardware looping. They implement looping in software instead. Some high-performance GPPs achieve nearly the same effect as hardware-supported zero-overhead looping by using branch prediction hardware. [10]

### Specialised addressing

DSP processors often support specialised addressing modes that are useful for common signal-processing operations and algorithms. Examples include modulo (circular) addressing, which is useful for implementing digital-filter delay lines, and bit-reversed addressing, which is useful for performing a fast Fourier transform. These highly specialised addressing modes are not often found on GPPs, which must rely on software to implement the same functionality. [10]

**Execution Time Predictability**

Aside from differences in the specific types of processing performed by DSP processors and GPPs, there are also differences in their performance requirements. In most non-DSP applications, performance requirements are typically given as a maximum average response time. That is, the performance requirements do not apply to every transaction, but only to the overall performance. In contrast, the most popular DSP applications, such as modems, are hard real-time applications. This means all processing must take place within some specified amount of time in every instance. This performance constraint requires programmers to determine exactly how much processing time each sample will require, or at least the worst-case time. It becomes critical if you attempt to use a high-performance GPP to perform real-time signal processing. Execution time predictability will probably be easy if you plan to use a low-cost GPP for real-time DSP tasks, because low-cost GPPs have relatively straightforward architectures and easy-to-predict execution times. Most realtime DSP applications require more horsepower than low-cost GPPs can provide however, so the choice is either a DSP processor or a high-performance GPP. [10]

## 2.2 Selection Criteria for DSP Processors

There are many factors that influence the choice of a DSP processor. The application that the DSP processor is intended for has the biggest influence on its choice. Satellite use requires specific device characteristics. These are discussed in the following subsections.

### 2.2.1 Power Consumption

Satellites operate off rechargeable batteries while the solar panels are in the dark. The batteries store a limited amount of power, so low power consumption of the devices onboard is very important to prolong battery life. Another advantage of lower power consumption is that less power is dissipated by the device as heat. This simplifies the thermal design of the satellite.

The supply voltage of the DSP processor has a big influence on the power consumption. Virtually all current DSP processors are fabricated using Complementary Metal-Oxide Semiconductor (CMOS) technology. Power consumption in CMOS devices is proportional to the the square of the supply voltage. This means that 3,3 V devices consume about 56 percent less power than 5 V devices. The disadvantage of a lower supply voltage is that devices become more susceptible to Single Event Upsets (SEUs), which is caused by radiation. This is due to the lower voltage threshold needed to change the state of the transistors. See Section 3.2 for more detail on radiation effects. [23]

Another way of reducing power consumption is to incorporate power management features on the processor. These features comprise control over the master clock frequency of the processor, and which parts of the processor receives the clock. This controls power consumption, because in CMOS devices energy consumption is linearly proportional to the clock frequency. A lower clock frequency will therefore result in less power being used.

DSP processors implement power management as instructions or registers which make the processor enter a sleep or idle mode. When it enters such a mode, the processor does nothing, and large parts of the processor are switched off. The processor is usually "woken" by an internal or external interrupt. It must be taken into account that it requires between two and a thousand instruction cycles to wake the processor, depending on the power-down mode. [23]

Another way to save power is by clock frequency control. When a processor's load does not require high speed operation, the clock frequency can be divided down. This is a compromise between full-speed operation and a sleep mode. To enable this, DSP processors enable the programmer to divide the clock speed by programmable factors, or to use a slow-speed clock. A considerable amount of power saving can be obtained with lower clock speeds when the device is not running full-speed.

The last option is to switch off peripherals of the device that are not used. By disabling the clock signal to them, more energy is saved.

### 2.2.2 Radiation Tolerance

All digital devices are affected by radiation. It is important to know how sensitive a device is to upsets in the space environment. The lifetime of a device in space depends on its radiation characteristics. Radiation hardened devices last much longer in space and are less susceptible to upsets, but are orders more expensive than Commercial Off The Shelf (COTS) devices. There has been a growing movement towards using COTS devices in the last few years, mostly due to their lower cost. This has led to more research going into their use in space.

Most devices undergo radiation testing on the ground before they are used aboard satellites or other spacecraft. This is to determine whether the device will comply with the planned lifetime of the spacecraft. In the case of SUNSAT, the DSP processor was flown without prior radiation testing. Various DSP processors have successfully been used in space, making them good candidates to use again. Satellites in Low Earth Orbit (LEO) environments do not require such high radiation tolerances as ones in higher orbits. This is discussed in more detail in Section 3.4.

### 2.2.3 Execution Speed

The speed at which a DSP processor must run to be able to perform a certain task is another criteria when choosing a DSP processor. Determining the performance of a DSP processor is not as simple as using the specified master clock speed. The clock speed determines the rate at which instructions are executed, and is not a good performance indicator. Each DSP processor family has a unique set of assembler instructions which are optimised to do certain mathematical calculations in the shortest time. The variation in the instruction sets and the implementation of them on the DSP processor make certain calculations faster on some DSP processors than on others.

Some of the performance indicators given in datasheets are Million Instructions Per Second (MIPS) and Million Floating-point Operations Per Second (MFLOPS). These figures give the maximum tempo at which instructions (MIPS) or floating-point operations (MFLOPS) can be executed under optimal conditions. The problem with these figures

is that the DSP processor does not operate under these optimal conditions continuously. Another factor is that these numbers do not give real performance figures, since performance depends on how much work is done per instruction. This is not reflected in the MIPS and MFLOPS ratings at all. The amount of work done per cycle depends on the processor architecture and its instruction set efficiency.

The shortcomings of clock-speed, MIPS and MFLOPS as performance indicators make it necessary to use benchmarking. DSP processor datasheets usually have benchmarking figures for certain mathematical calculations, which are useful to compare processors in that family. To compare different makes of DSP processors, it is safer to use independent benchmarks. The ideal is to generate benchmarks with the application software on each considered processor. This requires a lot of extra time and money however, because the software must be optimised for each processor. The other option is to use the results of an independent company, like Berkeley Design Technology Incorporated (BDTI). They have developed an independent, single-number measure of a processor's execution speed of DSP intensive applications. This benchmark, called the BDTImark2000, is published free of charge for various popular DSP processors. The advantage of this benchmark is that the single number gives a quick indication of the DSP processor's general performance. The disadvantage is that more specialised applications may not perform according to this benchmark because they rely heavily on a specific type of calculation. [10] [3]

### 2.2.4 Fixed- or Floating-Point DSP Processor

The most fundamental difference between fixed- and floating-point DSP processors is the numeric format. Fixed-point DSP processors represent numbers as integers or fractions between -1 and 1. Floating-point processors use integer or floating-point number representation. The differences between them are discussed below.

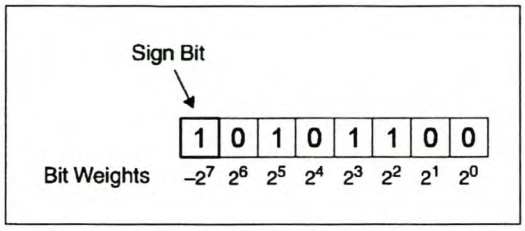



**Figure 2.2:** The integer number representation of an 8-bit value. [23]

As shown in Figure 2.2, integer number representation consists of a number of bits, each with a certain weight. The bit with the biggest weight is the sign bit, which counts negative. The value of the number is obtained by adding the weights of all the bits which are "1". For example the number from Figure 2.2 is calculated as follows:

Decimal value $= -2^7 + 2^5 + 2^3 + 2^2 = -128 + 32 + 8 + 4 = -84$

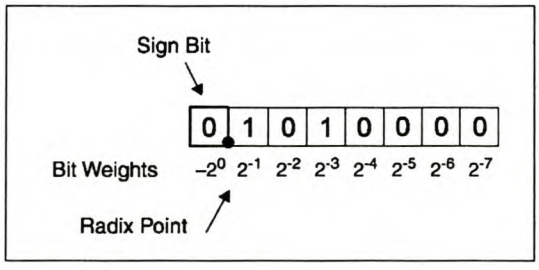



**Figure 2.3:** An example of a fixed-point number representation. [23]

As with integer number representation, the value of fixed-point numbers are obtained by adding the weights of all the bits which are "1". The number in Figure 2.3 is calculated in the following way:

Decimal Value $= 2^{-1} + 2^{-3} = 0,5 + 0,125 = 0,625$

As can be seen from the above two figures, integer and fixed-point number formats are very similar. The algorithms and hardware used to implement them on the DSP processor are virtually identical. The main difference between them is in how the results of multiplication operations are handled.

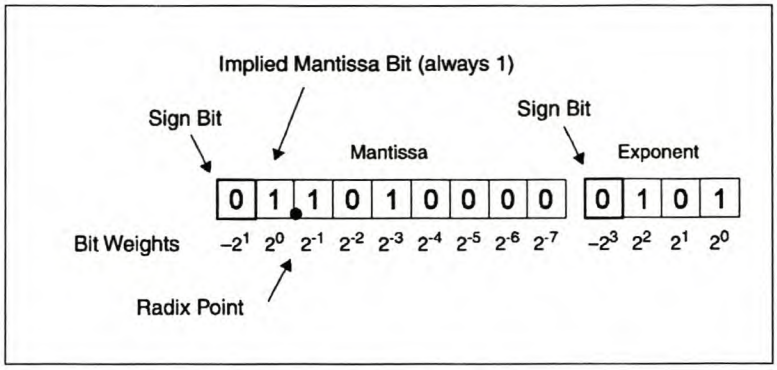



**Figure 2.4:** A simplified binary floating-point number representation, which comprises a mantissa and an exponent. [23]

Floating-point numbers consist of a mantissa and an exponent. The mantissa and exponent are calculated by adding the bit weights in each. The value of the number is given by $mantissa \times 2^{exponent}$, as is shown in the example below which calculates the value represented in Figure 2.4:

$\text{Mantissa} = 2^0 + 2^{-1} + 2^{-3} = 1 + 0,5 + 0,125 = 1,625$
$\text{Exponent} = 2^2 + 2^0 = 4 + 1 = 5$
$\text{Decimal Value} = 1,625 \times 2^5 = 52,0$

The precision of a number format refers to the error involved when rounding a value to fit into a certain number representation. For 16-bit fixed-point processors, the biggest error is $2^{-16} \approx 1,52587 \times 10^{-5}$. A floating-point processor's precision is given by die number of bits in the mantissa, including the implied bit. Most floating-point DSP processors use 32-bit data representation, of which 24 bits form the mantissa. This gives an error of $2^{-25} \approx 2,98023 \times 10^{-8}$. The more bits used, the smaller the error, and the higher the precision. This shows that the error of floating-point DSP processors is orders smaller compared to 16-bit fixed-point processors. 24-Bit fixed-point processors have a similar precision as floating-point processors. [23]

Dynamic range is defined as the ratio between the largest and the smallest numbers that can be represented by a numeric format. For example, the largest number a 32-bit fixed-point format can represent is $1,0 - 2^{-31}$ and the smallest is $2^{-31}$. The ratio between these numbers, which is the dynamic range, is approximately $2,147 \times 10^9$, or about 187 decibels (dB). A 32-bit floating-point format with a 24-bit mantissa and an 8-bit exponent

can represent numbers from approximately $5,877 \times 10^{-39}$ to $3,403 \times 10^{38}$. This gives a dynamic range of about $5,790 \times 10^{76}$, or 1535 dB. It is clear that floating-point formats have a clear advantage when it comes to dynamic range. [23]

Some DSP processors make use of extended precision, which refers to a processor using a higher precision format for calculations than its native data format provides. This involves using a wider data word for calculations and rounding off the result when it is stored in memory. The advantage of this is that while a series of calculations do not store the result, a very high precision and dynamic range is obtained.

Floating-point arithmetic can be emulated on fixed-point processors with special routines. This enables the high precision and dynamic range of a floating-point processor on a fixed-point one. Libraries with these routines are provided by some DSP processor manufacturers. The disadvantage of these routines is that they require many instructions of overhead to perform the emulation. This implies that floating-point emulation may only be practical when small portions of the code require the higher precision. [23]

Floating-point processors have the advantage of higher precision and dynamic range, but are usually more expensive than fixed-point processors. This is due to their more complex circuitry and wider program buses which result in a larger processor die. [23]

### 2.2.5 On-chip Features

Most DSP processors are available with various on-chip features, peripherals and peripheral interfaces. Most of these enable interfacing to other circuits or enhances the capabilities of the processor. Others like memory are essential for the operation of the processor. The peripherals include hardware such as serial ports, parallel ports, host ports, Analogue to Digital (A/D) and Digital to Analogue (D/A) converters, timers and JTAG ports. The functions of some of these features are briefly described below.

### RAM and ROM

DSP processors usually have a certain amount of RAM on-chip to store the program code and data. The amount of RAM on the processor depends on the application it is intended for. Another factor is that some processors have longer instruction words than others, and therefore require more memory to store the same amount of instructions. A smaller number of processors have on-chip ROM as well so no extra external devices are necessary to boot the processor. In some cases there is no memory on-chip, which makes the processor less expensive and lets the designer decide how much will be necessary.

### Serial Ports

Serial ports can be split into synchronous and asynchronous ports. Asynchronous ports operate at lower speeds and are typically used for RS-232 and RS-422 communications. Synchronous serial ports are capable of data rates as high as the clock speed of the processor. The advantage of serial ports over parallel ports is that they require as few as three or four interface pins, compared to parallel ports which are 8, 16 or 32 bits wide. Synchronous serial ports are often used to interface to A/D and D/A converters as well as other processors. The small and fast serial interface reduces the amount of pins needed on peripherals.

Not all synchronous serial interfaces are compatible with each other. Aspects like clocking through data, whether the high or low bit must be sent first, the width of the data, frame synchronisation and a few more determine compatibility. Some DSP processors have serial ports which can be configured to be compatible with different standards. This ensures that they are not limited to interfacing to certain devices only. [23]

### Parallel and Bit I/O Ports

Parallel ports enable fast data transmission by transferring 8, 16 or 32 bits at a time. This is done with a data bus and a strobe line to signal that new data is available on the bus. Some DSP processors have separate parallel buses and others use the main data bus as a parallel port.

Bit I/O ports are ports where individual bits can be programmed as input or output pins. These ports are handy for control purposes or for simple data transfers. Many processors have special instructions which test the state of a bit I/O pin. This allows for example a conditional jump to be performed depending on the state of the bit. Some processors have sophisticated bit I/O ports which check the bits for certain patterns and cause an interrupt or set flags accordingly. [23]

### Host Ports

Host ports are used to connect to another DSP processor or microprocessor. They allow data transfers between the two processors and some DSP processors allow control of the processor as well. In this case registers and memory can be read or written by the host.

### Debugging Interface

With systems becoming more complicated a debugging and emulation interface has become more important. Different types of debugging interfaces exist which allow a number of debugging features. A popular type of debugging is scan-based in-circuit emulation. This is implemented with dedicated hardware on the processor and a few interface pins. Debugging features such as downloading programs, examining and setting registers and memory, setting breakpoints and other functions are possible with this interface. All these features are available while the processor is installed in the target system.

Boundary-scan is a technique to read back the values on all the processor pins and force values to the pins as well. It uses a simple serial protocol to make this possible. This is a noninvasive method to test the processor and the devices connected to it in a system. The most popular interface used for boundary-scan implementations is IEEE standard 1149.1, also known as JTAG. The JTAG interface can also be used to communicate with on-chip debugging circuitry, and has become so popular that it is sometimes used for that purpose only.

### 2.2.6 Price and Availability

Another issue when choosing between DSP processors is their price. The price depends on many factors including the demand for the processor and competition from other makes of DSP processors. The process used to fabricate the processor die has a big effect as well. The smaller the die the more processors can be made at a time, and the less expensive the processor becomes.

The future availability of a DSP, or next generations being backwards compatible is important if the software developed is to be re-used in the future. Some DSP processors in the same family are pin-compatible and can use the same code with minor modifications. This allows easy upgrading to a faster or bigger processor if necessary.

## 2.3 DSP Applications on Satellites

The fixed-point DSP processor on SUNSAT was used for various functions on the satellite. The three main tasks it performed were implementation of different modulation schemes, image processing and compression, and as a repeater for ham radios. DSP processors are often used for modulation schemes because this allows adjusting the scheme, or using a different one altogether, after the satellite is launched. Image compression was implemented on SUNSAT to reduce the size of large images taken by the satellite. This helped to shorten the time needed to download an image through the available low-speed communications links. The repeater function allows ham radio amateurs to communicate with each other over longer distances than their radios usually allow. This is achieved by the satellite receiving the radio broadcasts and then transmitting them again. All radios in the footprint of the satellite are then able to receive the transmission.

DSP processors can also be used in star cameras on satellites. Star cameras determine the orientation of a satellite from images taken with a camera of the stars. The direction of the camera with reference to the satellite is known, so once it is determined in which direction the camera pointed when the photo was taken, the satellite orientation can be calculated. This requires image processing and recognition to map the photo onto a database of the most prominent stars.

The Attitude Determination and Control Systems (ADCS) of a satellite require a lot of computations to process the information from different sensors and adjust the satellite orientation accordingly. A DSP processor can be used to perform these computations.

## 2.4 Short List of DSP Processors

The criteria from Section 2.2 were used to compile a list of possible DSP processors for use on a next satellite. The tables list the most important features and specifications of DSP processors for space use. These are supply voltage, clock speed, MFLOPS, MIPS, fixed/floating-point, data width, power consumption, RAM, ROM and use in space. This information was retrieved from the processor datasheets and various web sites. The short list is tabled in Appendix A.

The makes of DSP processors considered were Analog Devices, Motorola, Lucent, Texas Instruments and Temic. These are the biggest suppliers of DSP processors, except for Temic who supply radiation hardened processors. The tables in the appendix are split up into these makes of DSP processors. In each table the different families of processors are distinguished by horizontal lines. The short list of processors was reduced to two possible DSP processors. The selection of these two processors is discussed in the following subsection.

### 2.4.1 Final Choice of DSP Processor

The DSP processor used on SUNSAT was a Motorola DSP56002 fixed-point processor. The need arose for a floating-point processor to provide more processing power for future satellites. This reduced the possible DSP processors from the short list to only a few. Analog Devices and Texas Instruments have the widest variety of floating-point processors and their processors have many features. This led to choosing the Analog Devices ADSP-2106x and Texas Instruments TMS320C3x families of processors as the best candidates for future use.

The Texas Instruments 320C30 and 320C31 processors have been successfully used on satellites, and the 320C30 has undergone radiation testing as well. No information could be found on whether any of the ADSP-2106x family of processors have been radiation tested or used in space before, and it was therefore assumed that there was no radiation data on them. The predecessor of the ADSP-2106x family, the ADSP-21020, was the first DSP processor to be used in space. A radiation hardened version of the processor, the TSC21020 is fabricated as well, but is too expensive for non-commercial applications. The ADSP-2106x processor family is built on the same ADSP-21000 core that the ADSP-21020 uses. The difference between them is that dual-ported on-chip SRAM and an I/O processor with a dedicated I/O bus was added to the ADSP-2106x family. [2]

Processor evaluation kits make testing a processor much easier because a board with basic peripherals and software are included in the package. This shortens the time to write new code and develop an application. Such kits are available for the TMS320C31 and ADSP-21061 processors. To save time developing new boards for testing the processors, the evaluation boards for these two processors were acquired.

The fact that there is no radiation information on the ADSP-2106x family of processors led to the decision to do radiation testing of the ADSP-21061 processor. Chapter 4 covers the radiation testing of the processor and the results of the tests.

New faster and more power-efficient DSP processors are released regularly. If a DSP processor with more processing power is needed in the future, one of these will have to be considered. With the SEU characteristics being so unpredictable for future fabrication methods, these processors will have to be radiation tested before they are used as well. Another option is using a general purpose processors which has DSP instructions. Currently they are not as fast as DSP processors, but in the near future they may be.

One important aspect of the DSP processors not covered by this thesis was writing software for them. The test programs that were written for this thesis were small and simple. Larger applications will have to be written to compare the ease of programming of these processors. Utilities such as debuggers and simulators play an important role in programming and have to be evaluated as well.

The features of the ADSP-21061 and TMS320C31 processors are listed in the following sections together with their evaluation kit details.

### 2.4.2 Features of the ADSP-21061 DSP Processor

The basic features and functions of the ADSP-21061 DSP processor are listed below. A diagram of the processor core follows at the end of the subsection.

Summary

- High Performance Signal Computer for Speech, Sound, Graphics and Imaging Applications
- Super Harvard Architecture Computer (SHARC) – Four Independent Buses for Dual Data, Instructions, and I/O
- 32-Bit IEEE Floating-Point Computation Units – Multiplier, ALU and Shifter
- 1 Megabit On-Chip SRAM Memory and Integrated I/O Peripherals – A Complete System-On-A-Chip
- Integrated Multiprocessing Features

Key Features

- 50 MIPS, 20 ns Instruction Rate, Single-Cycle Instruction Execution
- 120 MFLOPS Peak, 80 MFLOPS Sustained Performance
- Dual Data Address Generators with Modulo and Bit- Reverse Addressing
- Efficient Program Sequencing with Zero-Overhead Looping: Single-Cycle Loop Setup
- IEEE JTAG Standard 1149.1 Test Access Port and On-Chip Emulation
- 240-Lead MQFP Package or 225-Ball Plastic Ball Grid Array (PBGA)
- Pin-Compatible with ADSP-21060 (4 Mbit) and ADSP-21062 (2 Mbit)
- Flexible Data Formats and 40-Bit Extended Precision 32-Bit Single-Precision and 40-Bit Extended-Precision IEEE Floating-Point Data Formats
- 32-Bit Fixed-Point Data Format, Integer and Fractional, with 80-Bit Accumulators

Parallel Computations

- Single-Cycle Multiply and ALU Operations in Parallel with Dual Memory Read/Writes and Instruction Fetch
- Multiply with Add and Subtract for Accelerated FFT Butterfly Computation
- 1024-Point Complex FFT Benchmark: 0.37 ms (18,221 Cycles)

1 Megabit Configurable On-Chip SRAM

- Dual-Ported for Independent Access by Core Processor and DMA
- Configurable as 32K Words Data Memory (32-Bit), 16K Words Program Memory (48-Bit) or Combinations of Both Up to 1 Mbit

Off-Chip Memory Interfacing

- 4-Gigawords Addressable (32-Bit Address)
- Programmable Wait State Generation, Page-Mode DRAM Support

DMA Controller

- 6 DMA Channels
- Background DMA Transfers at 50 MHz, in Parallel with Full-Speed Processor Execution
- Performs Transfers Between ADSP-21061 Internal Memory and External Memory, External Peripherals, Host Processor, or Serial Ports

Host Processor Interface

- Efficient Interface to 16- and 32-Bit Microprocessors
- Host can Directly Read/Write ADSP-21061 Internal Memory

Multiprocessing

- Glueless Connection for Scalable DSP Multiprocessing Architecture
- Distributed On-Chip Bus Arbitration for Parallel Bus Connect of Up To Six ADSP-21061s Plus Host
- 300 Mbytes/s Transfer Rate Over Parallel Bus

Serial Ports

- Two 40 Mbit/s Synchronous Serial Ports
- Independent Transmit and Receive Functions
- 3- to 32-Bit Data Word Width
- $\mu$-Law/A-Law Hardware Companding
- TDM Multichannel Mode
- Multichannel Signaling Protocol

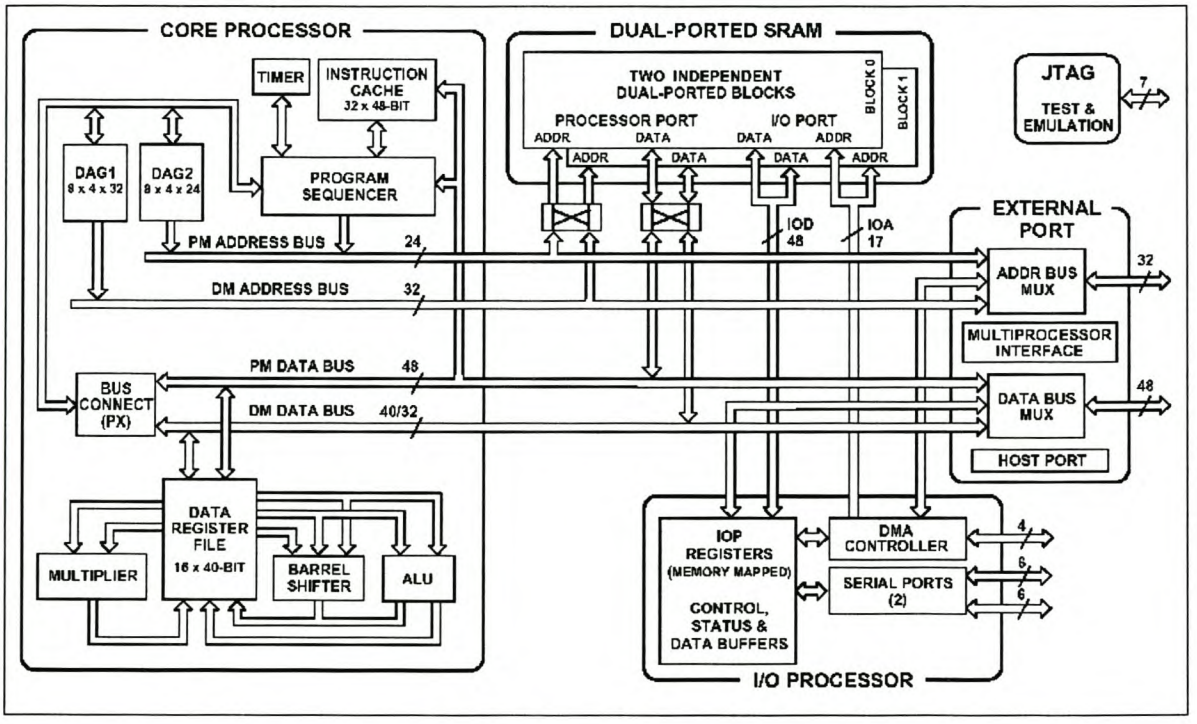



**Figure 2.5:** A block diagram of the ADSP-21061 DSP processor core. [2]

### 2.4.3 ADSP-21061 Evaluation Board

The SHARC EZ-KIT Lite uses the ADSP-21061, the newest member of the SHARC family of 32-bit DSP processors. Included with the SHARC EZ-KIT Lite are the following:

- Software Development Tools
- C Compiler with numerical C extensions
- Assembler, Linker, instruction-level Simulator
- C Source Level Debugger, and C Runtime Library
- A kernel for power-on self-test and host interface
- A host-based Windows 3.1 uploader/downloader to allow the user to perform data uploads and downloads from/to the target SHARC EZ-KIT Lite
- Diagnostic utilities to control program operation and access DSP memory
- EZ-KIT Lite Board
- RS-232 interface with UART
- AD1847 SoundPort 16-bit, full-duplex audio CODEC
- Line in and line out with stereo jacks
- Socketed EPROM, 128K x 8 (27C010) Firmware includes a power-on self-test with audio report and a RS-232 based boot-strap loader
- EZ-ICE emulation header

The SHARC EZ-KIT Lite CD-ROM includes the following documentation:

- SHARC EZ-KIT Lite Reference Manual
- ADSP-21000 Family Development Software Tools 3.3 Release Note
- ADSP-21000 Family Assembler Tools Manual
- ADSP-21000 Family C Tools Manual
- ADSP-21000 Family C Runtime Library Manual
- ADSP-2106x SHARC DSP User's Manual
- ADSP-2106x Family Data Sheets
- Digital Signal Applications Using the ADSP-21000 Family, Volume 1
- On-line Tutorial Block Diagram

### 2.4.4 Features of the TMS320C31 DSP Processor

The TMS320C31 DSP processor has the following key features and a block diagram of the processor is given at the end of the subsection.

- TMS320C31-50 (5 V)
  40-ns Instruction Cycle Time, 50 MFLOPS, 25 MIPS
- 32-Bit High-Performance CPU
- 16-/32-Bit Integer and 32/40-Bit Floating-Point Operations
- 32-Bit Instruction Word, 24-Bit Addresses
- Two 1K × 32-Bit Single-Cycle Dual-Access On-Chip RAM Blocks
- Boot-Program Loader
- On-Chip Memory-Mapped Peripherals:
  – One Serial Port
  – Two 32-Bit Timers
  – One-Channel Direct Memory Access (DMA) Co-processor for Concurrent I/O and CPU Operation
- Fabricated Using 0.6 mm Enhanced Performance Implanted CMOS (EPICE) Technology by Texas Instruments (TIE)
- 132-Pin Plastic Quad Flat Package (PQ Suffix)
- Eight Extended-Precision Registers
- Two Address Generators With Eight Auxiliary Registers and Two Auxiliary Register Arithmetic Units
- Two Low-Power Modes
- Two- and Three-Operand Instructions
- Parallel Arithmetic/Logic Unit (ALU) and Multiplier Execution in a Single Cycle
- Block-Repeat Capability
- Zero-Overhead Loops With Single-Cycle Branches
- Conditional Calls and Returns
- Interlocked Instructions for Multiprocessing Support
- Bus-Control Registers Configure Strobe-Control Wait-State Generation

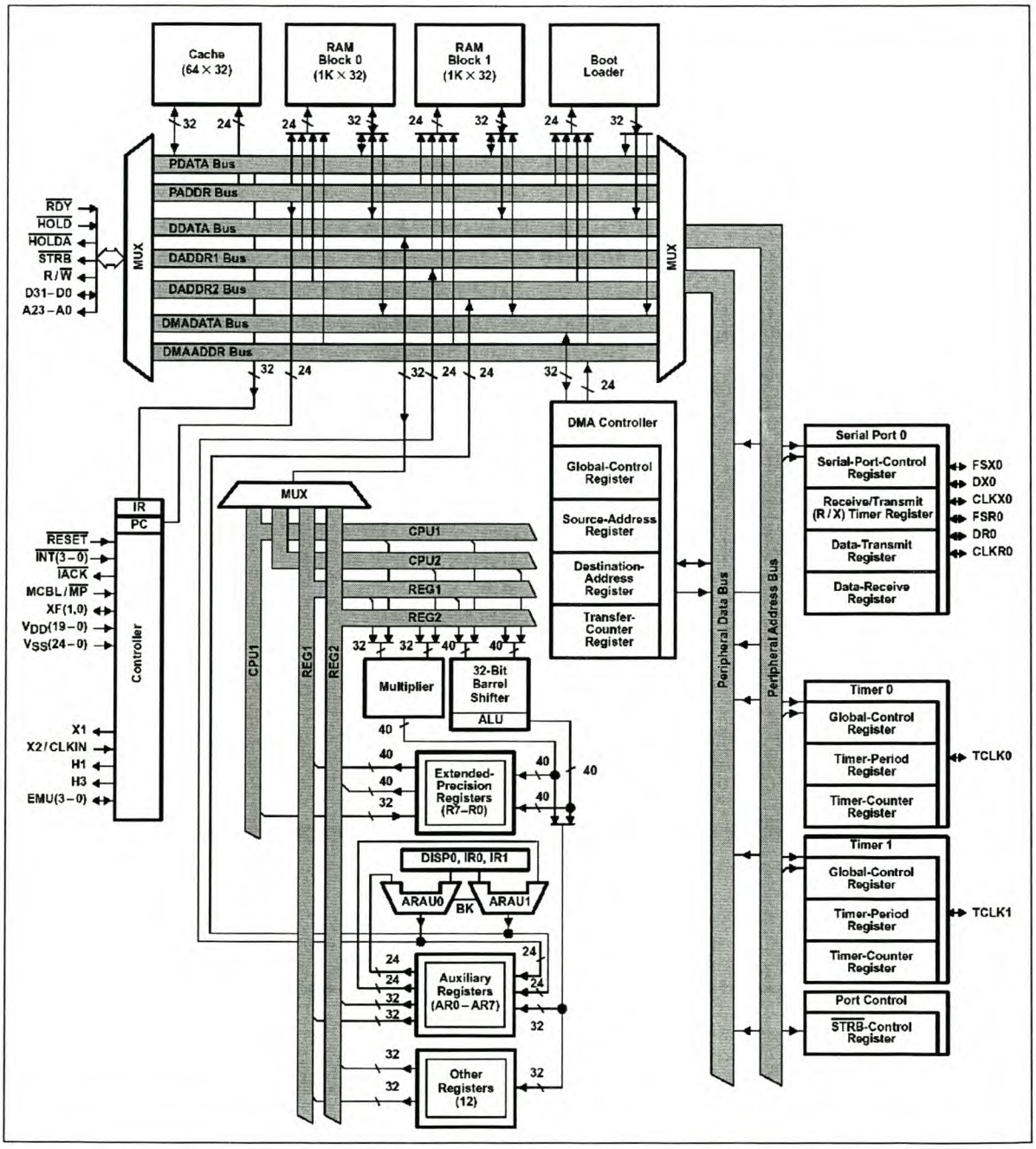



**Figure 2.6:** A functional block diagram of the TMS320C31 DSP processor. [35]

### 2.4.5 TMS320C31 Starter Kit

The TMDS3200031 Starter Kit for the TMS320C31 has these features:

- TMS320C31 floating point DSP
- 40 ns instruction cycle (50MFLOPS, 25 MIPS)
- TLC32040 Analogue Interface Circuit
- Standard or Enhanced Parallel Printer Port Interface
- Debugger
- Assembler
- Analogue data input and output via the TLC32040 analogue interface circuit:
- variable rate A/D and D/A converters
- A/D converter sample and hold is 14 bit impulse accurate at 20KHz with bypassable switched-capacitor anti-aliasing filter
- Switched capacitor low-pass filtered D/A converter output
- Access to 'C31 serial port and analogue power through jumper block
- Standard RCA plug connectors for analogue input and output
- XDS 510 emulator support (header not installed)
- I/O expansion bus for daughter-boards
- Tri-colour LED power indicator (colour change through PWM phase differential of 'C31 timers)

# Chapter 3

# Radiation Effects

The chapter starts with a description of the space radiation environment, covering the sources of the particles that affect satellites. The effects that these particles have on semiconductors are discussed next, followed by a section on how to reduce these effects. The chapter ends with a summary of the four orbits a satellite typically travels in, and which radiation types play the largest role in those orbits.

## 3.1 Space Radiation Environment

The space environment can be split up into two parts, the transient radiation environment and the trapped radiation environment. The transient environment consists of particles that tend to have high energies and pass by quickly. The trapped radiation environment comprises particles that are trapped in belts around the earth and stay there for longer periods. Depending on the orbit of a satellite, these environments play a larger or smaller role in the effects on satellites.

### 3.1.1 Transient Radiation Environment

There are different transient sources of radiation, of which some are solar flares, solar wind and galactic cosmic radiation. These are described in more detail below.

#### Solar Flares

Solar flares are caused by magnetic disruptions of the solar photosphere. This causes a variety of particle types and energies to erupt from the sun into space. The energy of protons may reach a few hundred MeV and that of heavy ions range from 10's of MeV to 100's of GeV. The kinetic energy of a particle is usually expressed in terms of Joule, but it is more comfortable to express these energies in MeV ($1\ eV = 1,6 \times 10^{-19}$ J is the energy of a proton accelerated over a potential difference of 1 volt). The average solar cycle lasts 11 years, which can be divided into four inactive years (solar minimum) and seven active years (solar maximum). The sunspot numbers in Figure 3.1 shows the solar cycle for the last 100 years. Large solar flare events mainly occur during the solar maximum, and may last for several hours to a few days. [31]

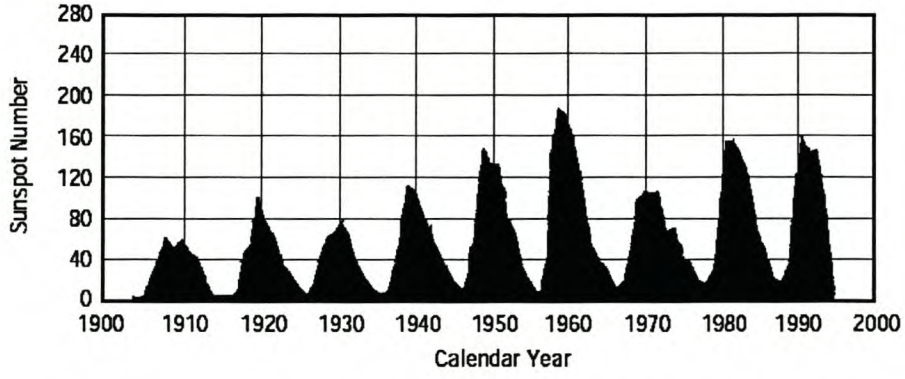



**Figure 3.1:** The yearly mean sunspot numbers for 1900 to 1992. The variations in the sunspot numbers indicate the times of solar minimum and maximum of the solar cycle. [18]

#### Solar Wind

The solar wind streaming from the sun is composed primarily of low energy protons and electrons, which are ejected from the surface of the sun. Due to the low energy of these particles compared to those from solar flares, the effect of the solar wind is often ignored. The travel time for the particles to reach the earth is in the order of 3 days. Unlike solar flares, the solar wind does not change rapidly over time. [20]

#### Galactic Cosmic Radiation (GCR)

Galactic cosmic ray particles originate outside the solar system from stars, supernovas, the galaxy centre and other energetic events. The particles consist of electrons, protons and heavy ions. The flux levels of these particles are low, but their energy levels are high (10's of MeV to 100's of GeV). These high energy levels cause intense ionisation when the particles pass through matter, posing a threat to sensitive electronics. [31] [9]

### 3.1.2 Trapped Radiation Environment

Energetic electrons and ions are magnetically trapped around the earth forming radiation belts, also known as the "Van Allen belts". Usually there are three belts, two electron belts and one proton belt. They extend from approximately 500 km to 76 000 km above the earth's surface [20]. The radiation belts consist mainly of electrons of up to a few MeV energy and protons of several hundred MeV energy. The proton intensities range from 1 proton/cm$^2$/sec to $1 \times 10^5$ protons/cm$^2$/sec [31]. Figure 3.2 illustrates how the particles spiral back and forth along the earth's magnetic field lines in the radiation belts.

Over the South Atlantic an area of enhanced radiation, known as the "South Atlantic Anomaly" (SAA), is caused by the offset and tilt of the geomagnetic axis with respect to the earth's rotation axis. This brings part of the inner radiation belt to lower altitudes (Figure 3.3). The position of the SAA over earth is shown in Figure 3.4. On the other side of the earth over Southeast Asia the opposite effect occurs, here the radiation belts lie at a higher altitude. [20]

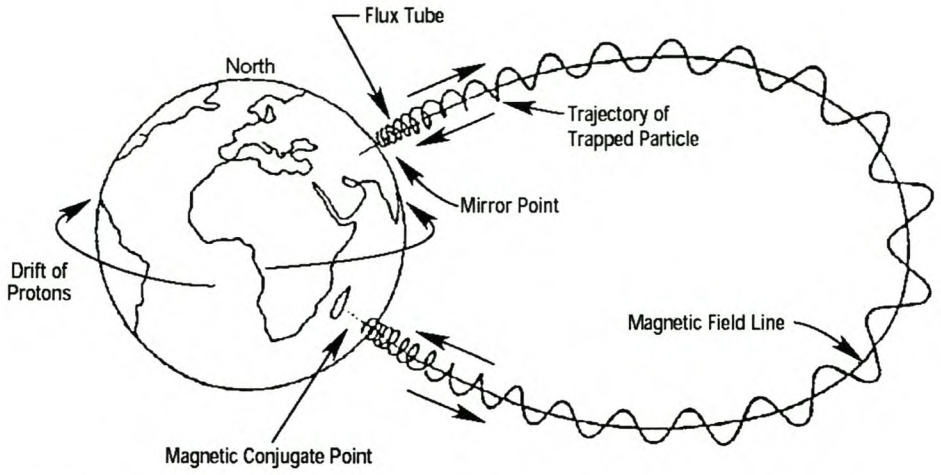



**Figure 3.2:** An illustration of the motion of trapped particles in the earth's magnetic field. [18]

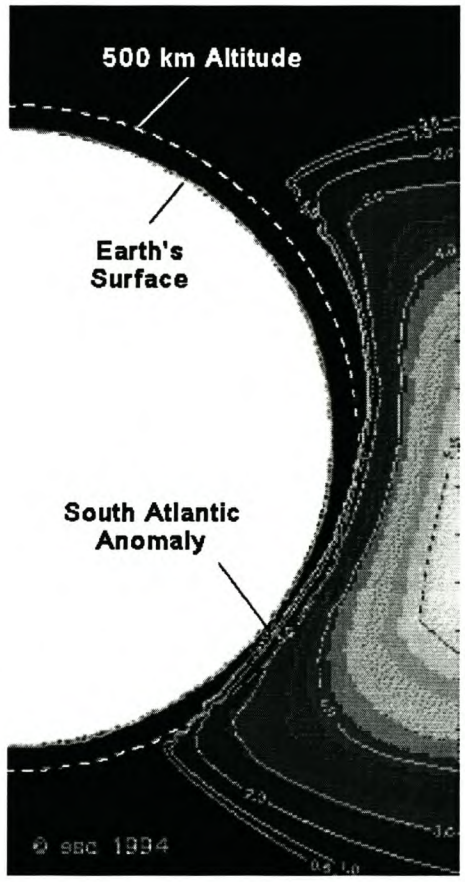
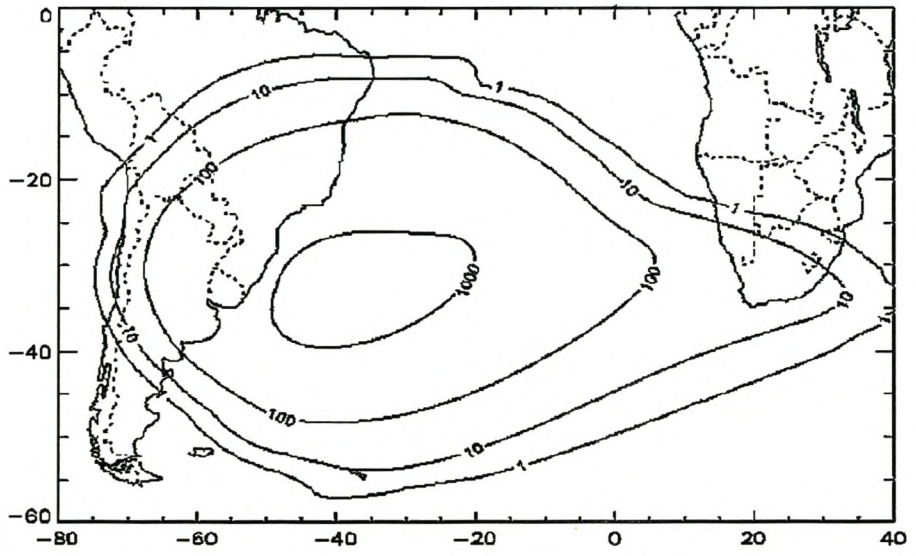



**Figure 3.3:** The proton flux variation with altitude, showing how the radiation belts drop lower in altitude at the SAA. [18]




500 km altitude, > 50 MeV proton fluxes, AP8MIN model

**Figure 3.4:** Particle fluxes at 500 km altitude, showing the position of the SAA over earth. [18]

## 3.2 Effects of Radiation on Semiconductors

The different particles in space can influence semiconductors in different ways. These effects together with the changes in semiconductor technology are described in this section.

### 3.2.1 Total Ionising Dose Effect

Total Ionising Dose (TID) is the accumulation of ionising radiation over time. The steady accumulation of ionisation over the life of an integrated circuit causes threshold shifts, increased device power consumption, timing changes, decreased functionality and eventually device failure.

Total dose creates electron-hole pairs in the silicon dioxide layers of MOS devices. As these begin to recombine, they create photocurrents and changes in the threshold voltage that make n-channel devices easier to turn on and p-channel devices more difficult to turn on. Even though some recovery and self-healing takes place in the device, the change is essentially permanent. Some holes created during ionising pulses are trapped at defect centres near the silicon/silicon oxide interface. Charges induced in the device create a high enough field across the gate oxide to cause the gate oxide to fail, or sufficient carriers are generated in the gate oxide itself to cause failure. [5] [20]

### 3.2.2 Transient Dose Effect

A Transient Dose is a high-level pulse of radiation, typical in a nuclear burst, which generates photocurrents in all semiconductor regions. This pulse creates sudden, immediate effects such as changes in logic states, corruption of a memory cell's content, or circuit ringing. If the pulse is large enough, permanent damage may occur. Transient doses can also cause junction breakdown or trigger latchup, destroying the device. [5]

### 3.2.3 Neutron Effect

When neutrons strike a semiconductor chip, they can displace atoms within the crystal lattice structure (Figure 3.5). Silicon devices begin exhibiting changes in their electrical characteristics at levels of $1 \times 10^{10}$ to $1 \times 10^{11}$ neutrons/cm$^2$. Because bipolar devices are minority carrier types, neutron radiation affects them more than Metal Oxide Semiconductor (MOS) devices. In bipolar integrated circuits, the base transit time and width are the main physical parameters affected. Therefore neutron radiation significantly reduces gain in bipolar devices. MOS devices are not usually affected until levels of $1 \times 10^{15}$ neutrons/cm$^2$ are reached. [5]

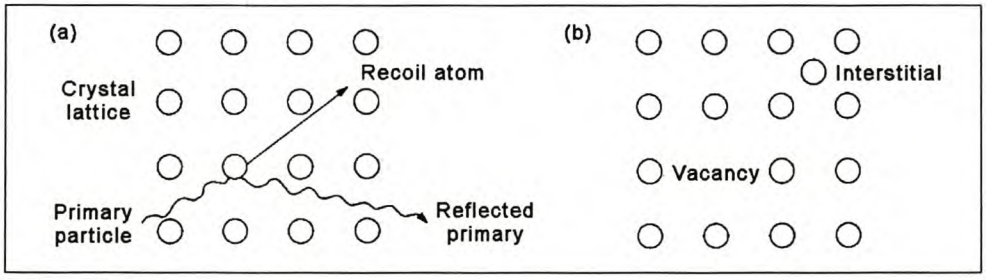



**Figure 3.5:** A schematic of atomic displacement in a crystalline solid : **(a)** atomic displacement event and **(b)** simple radiation-induced defects (vacancy and interstitial). [28]

### 3.2.4 Displacement Effect

Displacement damage occurs when protons or electrons with high enough energy ($\geq$ 1 MeV for electrons, $\geq 10^2$ keV for protons) interact with a target atom in a crystalline lattice. Damage occurs in such lattices as a result of elastic two-body collisions, electronic interactions with charged particles, and nuclear interactions (Figure 3.5). Since silicon and many other materials have thresholds of about 25 eV for the creation of vacancy/interstitial pairs, considerable damage can be done by both primary particles and recoiling target atoms. [28]

The billiard ball-type collisions and electronic excitations take place for incident charged particles (electrons and protons) at energies of 1 MeV or less. Above 8 MeV, protons can

also cause inelastic nuclear interactions in which energies considerably above 25 eV can be transferred to the lattice. The charged particles such as protons prefer to lose their energy through electronic interactions. A 10 MeV proton loses about 1000 times more energy as a result of electronic interactions than energy as a result of nuclear processes. Neutrons, carrying no electronic charge, can cause damage only by elastic or inelastic nuclear collisions. Charged particles can produce more damage in a localised area of a lattice because of their preference for losing energy in electronic interactions. Neutrons have a greater range in materials, since they are not charged, and produce damage throughout the material. [28]

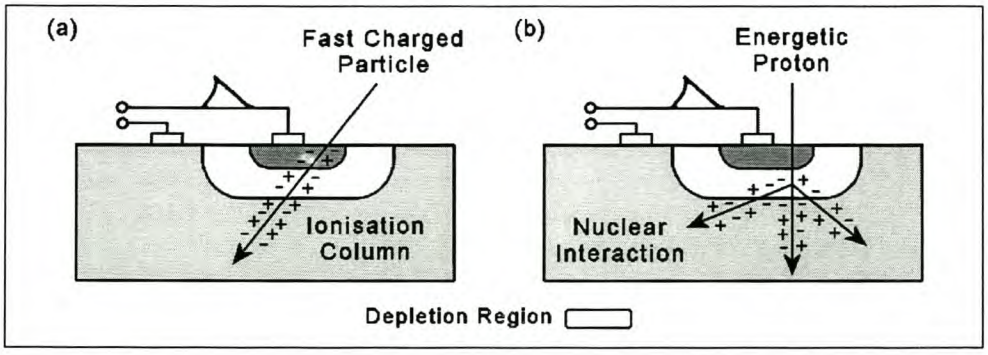



**Figure 3.6:** An illustration of how galactic cosmic rays deposit energy in an electronic device by **(a)** a proton or electron and **(b)** a proton or neutron. [25]

### 3.2.5 Single Event Effect (SEE)

SEEs are caused by energetic heavy ions that pass through materials and generate intense tracks of ionisation. If the ion passes through a sensitive part of a semiconductor chip, for example the silicon that stores a logical "bit", the free charge generated by the ion can be enough to change the logic state of the bit.

When an electron-hole pair is created by radiation, the electron in the valence band is excited across the bandgap into a conduction band state, leaving a hole behind in the valence band. If an electric field is present, the electrons are readily swept away as their mobility in silicon is much greater than that of the holes. The created electrons and holes are free to drift and diffuse in the material until they undergo recombination or are trapped, except for a small fraction of pairs that undergo recombination immediately.

Insulators such as silicon dioxide (gate and field oxides in transistors and integrated circuits) contain large densities of trapping centres at which the radiation-induced charge can reside for long periods of time. These charges generate internal space-charge electric fields that lead to voltage offsets or shifts in device operating characteristics. [28] [5]

SEEs are also caused when an energetic particle hits the nucleus of an atom. The nuclear interaction can produce spallation, which is the splitting of the nucleus. Spallation produces heavy debris which carries a sizable portion of the initial particle's energy. This debris generates ionisation, which has the same effects as described above. [8]

There are different SEEs that can occur. Some of these are single event upset, single event latchup, single event snapback, single event burnout and single event gate rupture. More information is given on these types of SEEs below.

### Single Event Upset (SEU)

SEUs occur when an ion deposits or depletes charge at a circuit node, causing a change in the state of a memory cell. In very sensitive devices, a single ion hit can also cause Multiple-Bit Upsets (MBUs) in adjacent memory cells. This type of event causes no permanent damage and the device can be reset or reprogrammed for correct function after such an event has occurred. [5] [16]

### Single Event Latchup (SEL)

SEL can occur in any semiconductor device which has a parasitic n-p-n-p path. A single heavy ion or high energy proton passing through either the base emitter junction of the parasitic n-p-n transistor, or the emitter-base junction of the p-n-p transistor can initiate regenerative action. This leads to excessive power supply current and loss of device functionality. The device can burnout unless the current is limited or the power to the device is reset. SEL is the biggest concern in bulk CMOS devices. A SEL is cleared by a power off-on reset of the device. [5] [16]

#### Single Event Snapback

This is also a regenerative current mechanism similar to SEL, but a device does not need to have a p-n-p structure. It can be triggered in a n-channel MOS transistor with large currents, such as IC output driver devices, by a single event hit-induced avalanche multiplication near the drain junction of the device. [5]

#### Single Event Burnout (SEB)

SEB may occur in power MOSFETs when the passage of a single heavy ion forward biases the thin body region under the source of the device. If the drain-to-source voltage of the device exceeds the local breakdown voltage of the parasitic bipolar, the device can burn out due to large currents and high local power dissipation. [5] [16]

#### Single Event Gate Rupture (SEGR)

SEGR has been observed due to heavy ion hits in power MOSFETs when a large bias is applied to the gate, leading to thermal breakdown and destruction of the gate oxide. It can also occur in nonvolatile memories such as EEPROMs during write or erase operations while a high voltage is applied to the gate. [5]

### 3.2.6 Influence of Semiconductor Technology

The feature size and supply voltage are two of the factors which have changed a lot through the years. In 1986 the process size and the supply voltage of the 8086 was 1,5 $\mu$m and 5 V, compared to 0,25 $\mu$m and 2 V for the more recent Pentium II processor. This scaling of the supply voltage and feature size is necessary to operate devices at higher clock speeds.

There has been concern that SEU, SEL and TID radiation hardness levels will decrease with smaller feature sizes and lower supply voltages. Studies have shown that certain

characteristics like TID have improved with scaling however. The SEU sensitivity of many commercial technologies seem to be close to their limit. This means that further scaling will not greatly influence the SEU behaviour. Devices that are less sensitive to SEUs than the current ones are still required for critical systems, which means that radiation hardened devices may have to be used. Latchup sensitivity has improved for many current technology circuits, but there are cases where circuits are still extremely vulnerable. Aspects like design, process and technology of fabrication methods change regularly, which makes it difficult to predict future TID, SEU and SEL properties of devices. Another factor is that new failure modes occur as devices are scaled down. [19]

## 3.3 Improving Device Radiation Tolerance

Different ways have been developed to improve the tolerance of semiconductors against all the effects mentioned in the above section. The best method is to use hardened semiconductor technologies, which are very expensive though. Another method used is shielding devices with metal or special materials. These are discussed below.

### 3.3.1 Hardened Integrated Circuit Technologies

The different technologies used to harden integrated circuits are junction isolation, dielectric isolation, silicon-on-sapphire and silicon-on-insulator. More details on these processes are given in this subsection.

#### Junction Isolation (JI)

Junction isolation is typically used for CMOS, and other unhardened bipolar designs. It consists of reverse biasing the junctions to isolate on-chip components from one another. Because this process is electrical, JI radiation tolerances may not be sufficient for circuits exposed to very high radiation levels. Junction Isolated processes are susceptible to latchup due to their parasitic PNPN structure. [5]

### Dielectric Isolation (DI)

Dielectric isolation is a step up from junction isolation. A thick layer of silicon dioxide is thermally grown between adjacent devices to provide component isolation. The oxide is constrained to grow only in chosen places on the wafer by using an oxidation mask. [5]

### Silicon-on-Sapphire (SOS)

Silicon-on-sapphire is a more complex form of dielectric isolation. A single-crystalline silicon film is grown over a sapphire substrate. The silicon island is doped to make a bipolar or FET transistor. Sapphire is a dielectric that has an inherently high tolerance to radiation. The sapphire protects the device against transient, neutron and single event effects. Leakage currents cannot flow between devices because the transistors are built on an insulating substrate. Therefore, guard rings that limit leakage current between transistors are unnecessary in SOS, and active devices can be packaged closer together. Also, there are no parasitic transistors to latch up, and there are no capacitances associated with SOS like there are with junction isolation reverse biased junctions. [5]

### Silicon-on-Insulator (SOI)

Silicon-on-insulator technology is very similar to the process used for silicon-on-sapphire devices. SOI and SOS have many of the same advantages. The main difference between them is the substrate used in each process. Silicon-on-insulator devices can take several forms, one of which is called Separation by Implanted Oxygen (SIMOX). In SIMOX, a high-current ion-implantation system is used to deposit a heavy concentration of oxygen molecules in a layer just below the wafer's surface. The wafer is then heated, and the oxygen forms a continuous $SiO_2$ layer beneath the silicon surface. The heating anneals the damage caused by the implant, and leaves a thin, high-quality layer of silicon on top of an insulating layer of $SiO_2$. This silicon is then used for device fabrication. Between active transistor areas, the silicon is etched away and replaced with oxide, which completely isolates the devices. This dielectric-isolation plane enables increased circuit speeds and radiation hardness. [5]

### 3.3.2 Shielding Components from Radiation

Shielding components with a high density material reduces the effect of total ionising dose. Protection against low energy particles causing SEUs is also achieved. Higher energy particles easily penetrate normal shielding however. Very thick shielding is needed to stop these particles.

One of the metals often used for the satellite structure due to its light weight is Aluminium (Al). If one wants to shield a component against protons of up to 120 MeV with Al for example, the Al thickness required can be read from the graph in Appendix C. The graph plots proton energy against the penetration depth of protons into Al and Brass. From the graph 120 MeV protons are stopped by about 5,1 cm of Al. To surround a component from all sides with 5,1 cm of Al is not practical. Brass would require less space, but the weight would be too much for use on a satellite.

## 3.4 Orbit Environments

Satellites travel in specific orbits, depending on the function of the satellite. The possible orbits are a low earth, highly elliptical, geostationary or polar orbit. The radiation environment of each of these orbits are discussed below.

### 3.4.1 Low Earth Orbit (LEO)

The most important characteristic of the environment encountered by satellites in Low Earth Orbits (LEOs) is that several times each day they pass through the proton and electron particles trapped in the Van Allen belts. The level of fluxes seen during these passes varies greatly with orbit inclination and altitude. The greatest inclination dependencies occur in the range between 0 and 30 degrees. For inclinations over 30 degrees the fluxes rise more gradually up to about 60 degrees. Over 60 degrees the inclination has little effect on the flux levels. The largest altitude variations occur between 200 and 600 km where large increases in flux levels are seen as the altitude rises. The flux increase

with increasing altitude is more gradual for altitudes over 600 km. The location of the peak fluxes depends on the energy of the particle. For trapped protons with E > 10 MeV, the peak is at about 4000 km. For normal geomagnetic and solar activity conditions, these proton flux levels drop gradually at altitudes above 4000 km. However, as discussed above, inflated proton levels for energies E > 10 MeV have been detected at these higher altitudes after large geomagnetic storms and solar flare events.

The amount of protection that the geomagnetic field provides a satellite from the cosmic ray and solar flare particles also depends on the inclination, and to a smaller degree the altitude of the orbit. As altitude increases, the exposure to cosmic ray and solar flare particles gradually increases. However, the effect that the inclination has on the exposure to these particles is much more important. As the inclination increases, a satellite spends more and more time in regions accessible to these particles. As the inclination reaches polar regions, it is outside the closed geomagnetic field lines and is fully exposed to cosmic ray and solar flare particles for a significant portion of the orbit.

Under normal magnetic conditions satellites with inclinations below 45 degrees will be completely shielded from solar flare protons. During large solar events the pressure on the magnetosphere causes the magnetic field lines to compress, resulting in solar flare and cosmic ray particles reaching otherwise unattainable altitudes and inclinations. The same can apply to cosmic ray particles during large magnetic storms. [31]

### 3.4.2 Highly Elliptical Orbit (HEO)

Highly elliptical orbits (HEOs) are similar to LEOs in that they pass through the Van Allen belts each day. HEOs have long exposures to the cosmic ray and solar flare environments for all inclinations, because of their high apogee altitude (greater than about 30 000 km). The levels of trapped proton fluxes that HEOs encounter depend on the perigee position of the orbit including altitude, latitude, and longitude. If this position drifts during the course of the mission, the degree of drift must be taken into account when predicting proton flux levels. [31]

### 3.4.3 Geostationary Orbit (GEO)

At geostationary altitudes, the only trapped protons that are present are below energy levels necessary to cause SEEs. However, GEOs are almost fully exposed to galactic cosmic ray and solar flare particles. Protons below about 40-50 MeV are usually geomagnetically attenuated. This attenuation breaks down during solar flare events and geomagnetic storms however. Field lines that cross the equator at about 7 earth radii during normal conditions can be compressed down to about 4 earth radii during these events. As a result, particles that were previously deflected have access to much lower latitudes and altitudes. [31]

### 3.4.4 Polar Orbit

Polar orbits are generally less than 1000 km in altitude, with inclinations above 80 degrees. They encounter the inner proton and electron belts in the form of the South Atlantic Anomaly, as well as the outer electron belt where the geomagnetic field lines bring it to low altitudes at "auroral" latitudes above 50 degrees. On the high-latitude parts of the orbit a spacecraft is exposed to almost unattenuated fluxes of cosmic rays and energetic solar particles. At low latitudes, geomagnetic shielding considerably reduces these fluxes. [31]

# Chapter 4

# Radiation Testing

The purpose of doing radiation tests on devices is to determine how much they are influenced by radiation. The various effects that radiation has on semiconductor devices are discussed in Chapter 3, Section 3.2. The two most important parameters are the susceptibility to single event effects (SEEs) and the total dose of radiation the device can absorb. Under SEEs, Single Event Upset (SEU) and Single Event Latchup (SEL) are the most important effects tested for. SEUs cause faults in the processor's operation and upsets in stored instructions and data. SELs induce a high current state, which can burn out the device if the power is not switched off. The total dose rating is a measure of how much radiation can accumulate in the device before it fails.

This chapter first looks at how radiation tests have been done previously, and then the choice of tests for the ADSP-21061 processor. The operation and layout of the National Accelerator Centre (NAC) where the tests were done is discussed afterwards. The next section covers all the details of the tests like hardware, software and procedure of testing. The results of the SEU testing and recommendations for future tests are given at the end.

## 4.1 Previously Reported Processor Radiation Tests

The two particles that cause the most SEUs in space are heavy ions and protons. The purpose of radiation testing is to use one of these two types of particles to simulate the effects found in the space environment. Flux rates in lower orbits are quite low, which results in very few upsets per day. Higher flux rates are used for testing so that the tests can be done in a few hours instead of days. The different particles, flux rates and fluences that have been used are discussed in the following subsection.

Testing a device's radiation hardness involves running programs on the processor while it is irradiated. Any incorrect functioning during this process is logged and used for cross-section calculations. Different parts of a processor are usually tested separately, for example the Cache, ALU and Memory of a processor. This provides a worst-case figure for upsets and gives an indication of which parts of the processor are the most susceptible to radiation. The figure is higher for these tests than for typical applications because a certain processor part will probably be used less than it is during the test. Details on the tests performed are given in the subsections that follow.

The last part of this section covers the different ways that have been used to detect when an upset occurs in the processor tested. These methods are coupled with the type of test programs that were run.

### 4.1.1 Radiation Used for Previous SEU Tests

The reasons for using ion and/or proton particles for SEU testing and which types were used are discussed below. This information was used to determine which energies, fluxes and fluences to use for irradiating the ADSP-21061 DSP processor.

Protons are used to determine device cross sections which are useful to predict upset rates for protons in the South Atlantic Anomaly or solar flare protons [26]. The proton energy, flux, fluence and test programs used for previous tests are listed in chronological order in Table 4.1.

Proton irradiation has a few advantages. Tests can be performed in air rather than vacuum, which makes setting up and changing the configuration much easier and faster. The lids of components do not have to be taken off, which is very difficult with most components. This is required for ion tests because ions do not have the same penetration depth as protons. Recent research has shown that for LEOs, irradiation with 200 MeV protons to $10^{10}$ protons/cm$^2$ also covers a majority of SEEs due to heavy ions [26]. The field size of proton beams allow screening of whole circuit boards at a time. [15]

Testing with heavy ions is used to determine the radiation sensitivity of devices in higher orbits where ions play a big role in causing upsets. This entails using a range of heavy ions with different energies. The ions used range from C-12 to Xe-609. Table 4.2 contains a summary of the types of ions used, and tests run, for previous SEU characterisation of devices. The Linear Energy Transfer (LET) unit is a measure of the amount of energy that the ions carry. A graph of LET against cross section is used to predict upset rates for different orbits in space.

**Table 4.1:** A summary of the proton beams used and test programs run on different processors during previous proton SEU tests.

| Year | Processor(s) | Proton Energy (MeV) | Flux ($p/cm^2/s$) | Fluence ($p/cm^2$) | Tests Done |
|---|---|---|---|---|---|
| 1992 | ADSP-2100A | 200<br>500<br>800 | | $5,3 \times 10^{11}$<br>$5,6 \times 10^{10}$<br>$2,0 \times 10^{10}$ | Register,<br>Memory |
| 1992 | R3000 | 13, 24, 40, 50, 60 | $10^7 - 10^{10}$ | | Register,<br>CPU,<br>Application |
| 1996 | MC68020,<br>MC68882,<br>TMS320C25 | 50, 100, 200<br><br>30, 50, 75, 100, 200 | | $10^9 - 10^{11}$ | Register,<br>ALU,<br>Memory |
| 1996 | 80386,<br>80486 | 26.6, 38.2, 63 | $10^{10}$ | $10^8$ | ALU,<br>DMA,<br>Addressing |
| 1999 | Pentium MMX,<br>Pentium II | 31, 61.5, 200 | $10^5 - 4 \times 10^7$ | $10^{10}$ | Register,<br>ALU + FPU,<br>Cache,<br>Memory |

**Table 4.2:** A summary of the ion beams used and test programs run during previous ion SEU tests.

| Year | Processor(s) | Ions used | Energy (MeV) | LET (MeV/ (mg/cm$^2$)) | Flux (particles/ cm$^2$/sec) | Tests Done |
|---|---|---|---|---|---|---|
| 1992 | ADSP2100A | C–12 | 84 | 1,7 | | Register |
| | | F–19 | 112 | 4,0 | | |
| | | Cl–35 | 153 | 12,7 | | |
| | | Ni–58 | 182 | 27,2 | | |
| | | I–127 | 217 | 53,0 | | |
| 1996 | 80386, 80486 | F–19 | 139 | 3,4 | $1 \times 10^3$– $2 \times 10^4$ | Operational |
| | | Si–28 | 190 | 7,79 | | |
| | | Ti–47 | 182 | 20,1 | | |
| | | Ni–58 | 258 | 26,8 | | |
| | | Br–75 | 285 | 37,2 | | |
| | | I–127 | 311 | 59,6 | | |
| | | Au–197 | 323 | 80,9 | | |
| 1996 | ADSP2100, DSP32C, TMS320C30 | N–70 | | | | Register, ALU, Cache, Program execution |
| | | Ne–90 | | | | |
| | | Ar–190 | | | | |
| | | Cu–290 | | | | |
| | | Kr–380 | | | | |
| 1998 | 80C186, 80C286, SMJ320C30, SMJ320C40 | N–70 | | | | Register, ALU, Sequencing, Bus unit |
| | | Ne–90 | | | | |
| | | Ar–175 | | | | |
| | | Cu–283 | | | | |
| | | Kr–366 | | | | |
| | | Xe–609 | | | | |

### 4.1.2 Tests Run on Processors for Previous Tests

For the previous tests the main functional parts of a processor were identified before tests were conducted. The different test programs were then used to test those functional parts for SEUs. As can be seen in Tables 4.1 and 4.2, the main components that were tested are the registers, ALU, cache, memory and program counter. In some cases an operational test was done as well, where the target application was run during irradiation to determine what effects upsets had on the system.

Testing of the program counter was done by having two separate blocks of NOP operations and then jumping from the one to the other. If at any time the program counter jumped outside these two blocks a program counter upset was recorded.

The ALU tests performed a range of logical and arithmetic operations on the processor and compared the result with the correct one. If any of the operations were upset the result of the calculations would show it. This was done in a loop until enough data was gathered.

The memory, register and cache tests were done by writing either 55 hex (01010101 binary) or AA hex (the inverse of 55 hex) to all available locations. Throughout the test the values were read back and compared with the original value. Any upsets were logged by the test system.

### 4.1.3 Test Setups for Previous Tests

Previous setups for SEU testing used one of three methods to monitor the devices and log all upsets and errors. The "golden chip" method makes use of a second processor which is not irradiated. The two processors simultaneously run the same software, and any differences between them are recorded. The recordings are used to determine what type of error occurred. A variation of this method, the "virtual golden chip" method, records the outputs of the processor before irradiation. A special logic analyser is used to do this. When the device is irradiated the recorded pattern is compared with the current one, and all differences logged.

Another way to determine upsets is to connect the device to a computer. The computer controls the test type and checks the feedback from the device. The computer's parallel and serial port can be used for such functions.

The last method used was for the Pentium MMX and Pentium II processors. They were tested in Personal Computers (PCs), each with a hard drive, screen, keyboard and mouse. For the first part of these tests a DOS-based program was run which sent the results to the screen. The other tests were run under Windows NT, in which case a "blue screen" indicated an upset.

In all cases the upset rates were kept low enough so that the monitor device could log upsets fast enough. Another reason was that if the upset rate was too high, the time lost when the irradiated device had to be reset could have a significant effect on the results.

## 4.2 Selection of Test Methods

The first objective was to find a suitable test facility to supply the same particles that were used for the previous tests. The only accelerator available was at the National Accelerator Centre (NAC). They agreed to help us and provided beam time to perform the SEU tests. Staff at the NAC helped with the calculations, setup and testing at the centre. The SEU testing was done in two different places at the NAC. This was due to problems experienced at the first test location. Section 4.3 describes the NAC function, facilities and layout in detail and Section 4.4 covers the test setup inside the NAC.

The test programs decided on were the sequencing (NOP), cache, ALU and memory tests. They cover the most important parts of the ADSP-21061 DSP processor. The core of the processor is the same as that of the ADSP-2100A, which has already been tested, except that the feature size of the ADSP-21061 is smaller. The NOP, cache and ALU tests would determine if the feature size affected the SEU tolerance of the processor. The 1 Mbit on-chip memory is very useful, but its sensitivity to upsets had to be determined before it could be considered for use in space. If it was too sensitive external memory would have to be used. The test hardware and software developed to perform these tests are discussed in Sections 4.4 and 4.5.

## 4.3 National Accelerator Centre (NAC)

This section supplies information on the functions of the NAC and the capabilities of the accelerators there. The layout of the centre is discussed and the two locations where the tests were performed are indicated. Details of the NAC are given in Appendix D.

### 4.3.1 The function of the NAC

The National Accelerator Centre (NAC) is a multidisciplinary research laboratory administered by the National Research Foundation (NRF), providing facilities for:

- Basic and applied research using particle beams
- Particle radiotherapy for the treatment of cancer
- The supply of accelerator-produced radioactive isotopes for nuclear medicine and research.

Activities are based around four sub-atomic particle accelerators. The large Separated Sector Cyclotron (SSC) accelerates protons to energies of 200 MeV, and heavier particles to much higher energies. Charged particles to be accelerated are supplied by one of two smaller injector cyclotrons, one providing intense beams of light ions, and the other beams of polarised light ions or heavy ions. The fourth accelerator at the NAC is a 6MV Van de Graaff electrostatic accelerator.

The NAC brings together scientists working in the physical, medical and biological sciences. The facilities provide opportunities for modern research, advanced education, the treatment of cancers, and the production of unique radioisotopes. [36]

### 4.3.2 Cyclotrons at the NAC

Three cyclotrons are operated at the National Accelerator Centre. The main machine, a large Separated Sector Cyclotron (SSC), is a variable-energy machine capable of accelerating protons up to a maximum energy of 200 MeV. Two smaller conventional Solid Pole Cyclotrons (SPC), SPC1 and SPC2, are used as injectors for the larger SSC. They each have a maximum extraction energy of 8 MeV for protons.

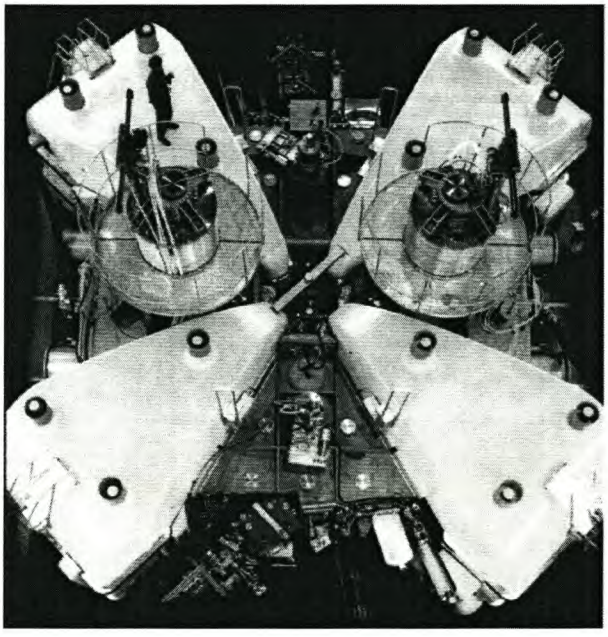


**Figure 4.1:** A photo of the Single Cycle Cyclotron (SCC) at the NAC. [36]

Each of the SPC1 and SPC2 cyclotrons have a specific function, and although they are very similar in construction, they vary in design. The main difference between the two machines is the source of ions. SPC1 has an internal ion source while SPC2 has two external ion sources with axial injection. The external ion sources are used because the sources are not compact enough to fit into the central region of the cyclotron.

SPC1, the first injector, uses an internal ion source to produce the intense beams of light ions required for radiotherapy and radioisotope production. The second injector, SPC2, provides heavy polarised ions which are used for experiments. With one of the ion sources a whole range of heavy ions such as carbon, argon, krypton, sodium, and other metals are accelerated. This ion source also serves as an alternative source of protons for therapy.

### 4.3.3 Particle Acceleration

It is possible to accelerate small charged particles to very high energies using a cyclotron or Van de Graaff accelerator. At the NAC the maximum energy reachable is 200 MeV for protons. The beam of particles can be directed onto a target material. Depending on the material, a nuclear reaction can be induced which leads to the formation of nuclei of a different kind. The formation processes of these formed nuclei are studied to learn more about the structure and properties of matter. [36]

### 4.3.4 How the Accelerator Works

In a cyclotron particles are accelerated over the potential difference between a so-called "dee" and "dummy dee". For two dees, four acceleration gaps are created. The dees are connected to a radio frequency (RF) generator with a frequency which ranges from just below 10 to 26 MHz. Ions are produced by an ion source in the centre of the cyclotron and are accelerated to the first dee, which has an opposite polarity to the particle. The maximum accelerating RF voltage for the NAC is 60 kV for SPC1 and SPC2, and 250 kV for the SCC. The voltage determines the final energy of the accelerated particles.

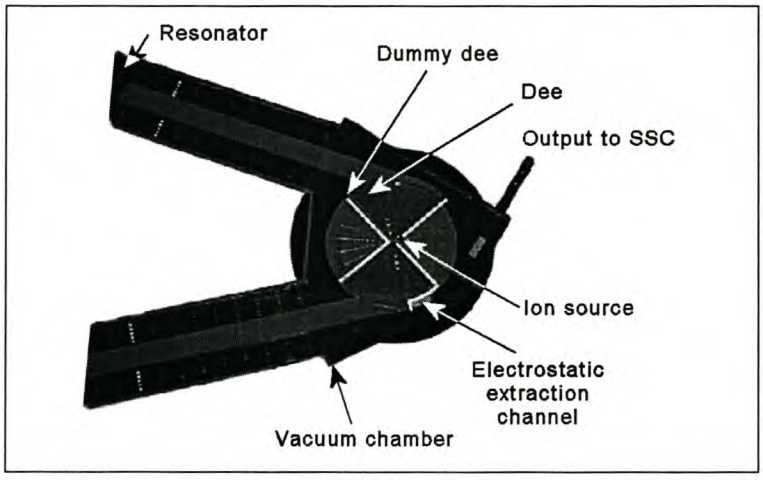



**Figure 4.2:** A diagram of a Solid Pole Cyclotron (SPC), showing the dee and dummy dee locations. [36]

Particle acceleration works on the principle that a charged particle, moving in a magnetic field, experiences a force perpendicular to the plane formed by the particle velocity vector and the magnetic field vector. With the right choice of magnetic field direction, the particle experiences a force directed to the centre of the machine. Within the dees the particle moves at a constant speed in a curve, due to the force it experiences. The polarity of the voltage on the dee is changed to positive and as it emerges from the dee it is accelerated over the gap towards the ground potential dummy dee. The moment the particle reaches the next acceleration gap, the facing dee is set to a negative polarity. This process is repeated, and the particle gains energy at every gap as it spirals out to the maximum radius, where it is extracted from the machine. This operation requires fine control of the magnetic fields and frequency. [36]

### 4.3.5 Layout of the NAC

The largest part of the NAC is situated together in a large block on the grounds. The Van de Graaff accelerator is located away from the other buildings. The layout of the main buildings of the NAC is pictured in Figure B.1. The two SPCs feeding the SSC can be seen to the bottom right of the picture. From the SSC the beam is steered to various vaults for experiments or to proton/neutron therapy stations. To the right of the figure the beam is used for radioisotope production as well. On the bottom the control room from where the cyclotrons and steering magnets are controlled can be seen.

Two areas of the NAC were used during the period that SEU tests were performed. The first tests were performed in a vacuum tank which is situated in a room next to the SSC. The location of the tank can be seen in Figure B.1 to the left of the SSC. The computer that ran the tests and its control board were set up in the data-taking room. The second series of test runs were done in the rightmost proton therapy station. The room in which the computer was stationed is above the therapy station on the figure in the physics laboratory.

## 4.4 Test Hardware

All the hardware that was used for the tests are discussed in this section. The hardware is split up into two parts. The first part covers the control hardware which was stationed in the data-taking room and physics laboratory. The second describes the circuits and setup in the radiation chambers, which were at the vacuum tank and therapy station. A schematic of all the circuits used for the tests and how they are connected together is given in Appendix B. It does not contain the same detail as the schematics given below, but gives an overview of the whole setup.

### 4.4.1 Data-taking Room and Physics Laboratory Hardware

This subsection describes the setup and circuits in the rooms from where the tests and radiation beam was controlled. The first part describes where everything was stationed in the rooms, and the second part the circuits and connectors that were used.

#### Setup in Data-taking Room and Physics Laboratory

The first SEU tests were done at the vacuum tank room. The computer and control board were set up in the data-taking room for these tests. The location of the data-taking room can be seen in Figure B.1. The computer and control board with power supply were placed on a table a few metres away from the patch board in the room. The patch board has coaxial cables running to a patch board underneath the vacuum tank. The rest of the equipment used were integrators and counters to measure the dose, a control board for adjusting the ladder and arms in the vacuum tank, and a computer to start and stop irradiation. This is all part of the permanent setup in the data-taking room.

The second series of experiments were done in one of the two proton therapy stations at the NAC. The station used is the rightmost one in Figure B.1. Just above this room is a physics laboratory in which the computer and control circuit was placed. The laboratory contains a patch board leading to the therapy station, and various computers to control the patient dose and display beam information. Figure 4.3 contains a photo of the laboratory

with the test computer to the left and the control circuit on the ground. To the right the racks with computers and other equipment can be seen. The control board is connected to the patch board in the middle.

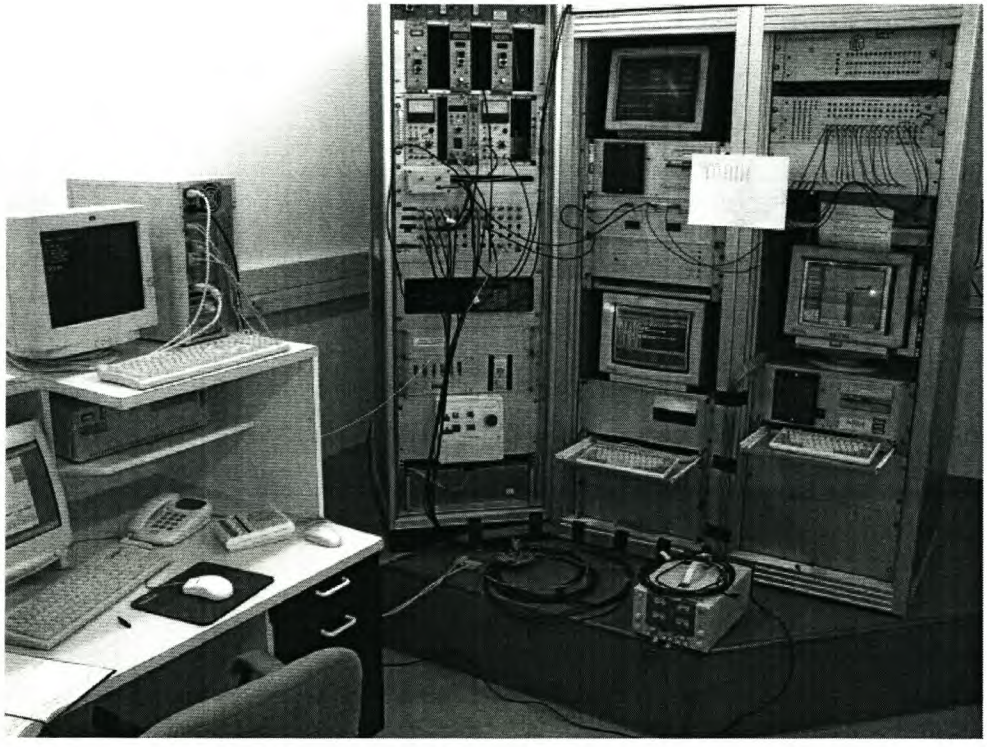


**Figure 4.3:** A photo of the physics laboratory setup. The position of the power supply, control board and test computer can be seen. The rack-mounted computers monitor the total dose and display beam information.

### Circuit and Connections in Data-taking Room and Physics Laboratory

A single circuit board was used to buffer the signals going to the other control board and to receive signals from there. The board is described in more detail below. The cables between the computer, control board and patch board comprised a parallel, serial and coaxial cable. The following signals pass through these cables:

- Temperature reading (to computer)
- Power switch (from computer)
- Board reset (from computer)
- Latchup interrupt (to computer)
- 3× Flag bits (to computer)
- Serial port (to and from computer)

The "temperature reading" cable carries a voltage which is proportional to the temperature of the DSP processor surface. This was used to determine if the processor got hot in a vacuum. The temperature reading cable was not used for the tests in the therapy station. The "power switch" signal turns the power to the evaluation board on and off. This was used to let the processor recover from a latchup. The processor was reset with the "board reset" line when it stopped responding. When latchup occurred the detection circuit sent a pulse across the "latchup interrupt" cable to stop the test on the computer. The three "flag bit" lines were used for all the tests except the memory test to send test information to the computer. The serial port lines are the three wires needed for basic serial communication. The circuits which generate and use these signals are described below.

All these signals, except the temperature reading one, are buffered by the control board. The temperature reading goes directly to a sampling card in the computer to convert the voltage into a temperature reading.

All digital signals passing to and from the radiation room were buffered. This was to ensure that no unnecessary load was placed on the DSP processor or other circuits when driving the 50 Ω coaxial cable. The buffer, as seen in Figure 4.4, is implemented with a 74HCT14 inverting Schmitt trigger and a 2N2222 transistor. The Schmitt triggers serve

as buffers for the circuits they connect to on both sides of the cable, and they improve noise immunity. The transistor provides the current to drive the coaxial cable from the output of the 74HCT14. The coaxial line was terminated with a 56 Ω resistor to match the impedance of the coaxial cable.

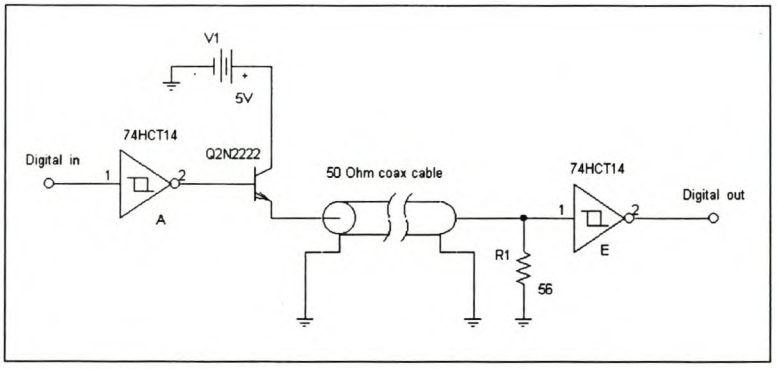



**Figure 4.4:** The digital buffer which ensures that the DSP processor and other circuits are not loaded by the coaxial cable.

The control board uses the push-pull configuration in Figure 4.5 to buffer the serial port signals. This is to ensure that the evaluation board serial driver is not loaded by the coaxial line.

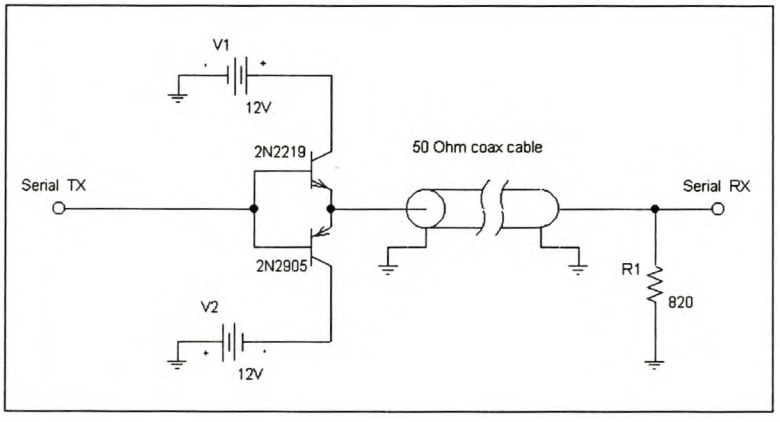



**Figure 4.5:** The buffer used for driving the coaxial cable from the RS232 serial port signals.

### 4.4.2 Vacuum Chamber and Therapy Station Hardware

The hardware used for the setup in the vacuum chamber room can be split into two parts. The first is the circuit for managing the signals and power supply to the DSP. The second is the shielding and mechanical fastening of the DSP processor evaluation board inside the vacuum tank. The general setups at the vacuum tank and therapy station are described first. The brass block and evaluation board configurations follow with a description of the connections and control board last.

#### Setup at Vacuum Tank

The vacuum tank is about 1,5 metres in diameter, with a height of 1 metre. The lid of the tank is hoisted off with a crane. A small platform on both sides of the tank gives access to work inside the tank. The radiation beam enters the tank through a pipe on the one side, and leaves through one on the other. Two pumps are used to create the vacuum, one to lower the pressure and another to create the near-vacuum.

Inside the vacuum tank there are two "arms" to which mountings can be made. The arms are connected to the middle of the tank, and can rotate around this point. One arm is slightly higher than the other so they can move past each other when turning. In the middle is a "ladder" which can move up and down. The ladder is used to change the energy of the radiation by moving metal blocks of different thicknesses into the path of the beam. The position of the arms as well as the height of the ladder is controlled from the data-taking room. The brass block was screwed onto a piece of tufnel, which was screwed onto a square spacing block. The spacing block screws directly onto one of the arms. The tufnel isolates the brass block from the arm assembly. The brass block is described in Section 4.4.2.

There are airtight ports around the outside of the vacuum tank. Attachments with a number of BNC-type connectors on both sides can be mounted on these ports. These BNC connecters were used for all electrical connections to the inside of the tank.

The control circuit lay next to the vacuum tank on a platform during the test. It is connected to the data-taking room via a patch board situated below the vacuum tank. The patch board has high quality 50 Ω coaxial cables between the boards, with female BNC connectors on both sides. The power supply for the board was placed on the same platform next to the board.

### Setup in Proton Therapy Station

The proton therapy station is usually used to treat patients. An automated chair is used to position the patients in the path of the beam. For our tests the chair was lowered beneath the floor level and a water tank moved into its place.

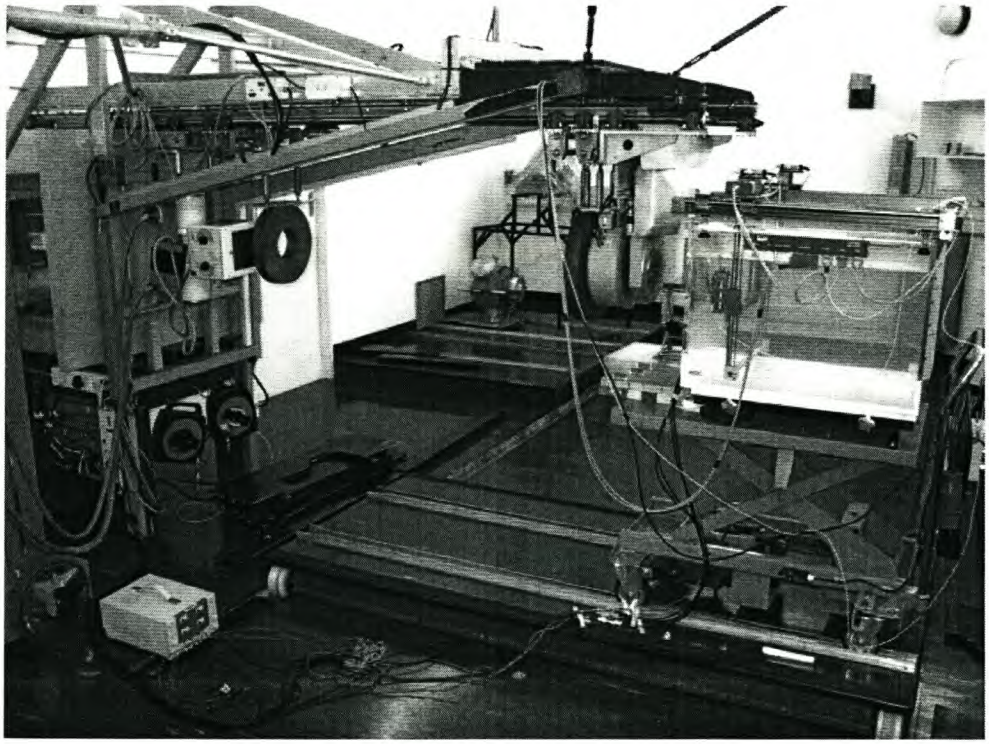


**Figure 4.6:** A photo of the setup inside the therapy station with the power supply, control circuit and water tank visible. The patch board can be seen to the left of the photo.

The water tank is used to calibrate the beam, and the trolley it moves around on was used to place the brass block on as well. The control circuit was placed on the ground next to the water tank with the power supply next to it. The patch board is located at

the bottom of the construction which holds the measuring devices and collimator in place. The setup can be seen in Figure 4.6. Figure 4.7 shows a closeup photo of the brass block between the water tank and the collimator.

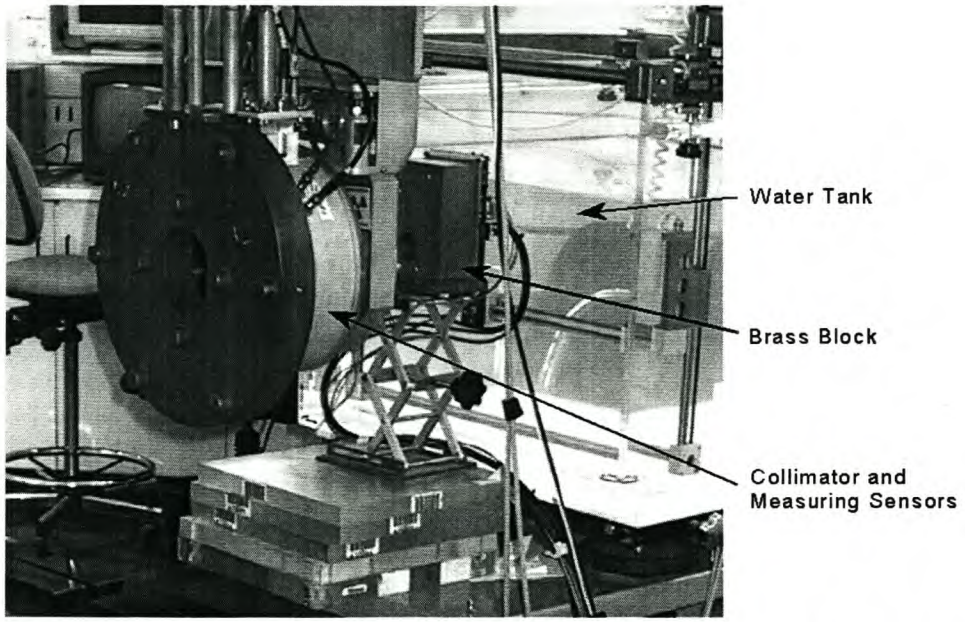



**Figure 4.7:** A closeup view of the brass block with evaluation board in front of the water tank.

### Brass Block and DSP Processor Evaluation Board

A block of brass was used to shield the DSP processor evaluation board from the proton beam. The block has dimensions of $95 \times 145 \times 55$ mm. From Figure C.1 it can be seen that 55 mm brass is enough to stop 200 MeV protons (Figure C.1 plots proton energy against proton penetration depth into brass and aluminium). The DSP processor is radiated through a 45 mm diameter hole in the block. A square piece of fibreglass board was glued to the brass block to mount the evaluation board on. The fibreglass provides electrical isolation from the brass block, which can charge up to high voltages when radiated. Four spacers fasten the evaluation board to the fibreglass. The board was mounted with the top facing away from the brass block. This was to make sure the copper heat slug in the DSP processor package did not influence the proton beam before it passed through the processor die. This orientation made connecting the test wires described below easier

as well. See Figure 4.8 for a photo of the brass block and evaluation board assembly. Another photo of the brass block without the board is available in Appendix B.

The SHARC EZ-KIT Lite evaluation board as described in Chapter 2, Section 2.4.1 was used for the radiation tests. The serial port on the board and three of the four general purpose flags of the processor interfaced the board to the test computer. A board reset signal was used to restart the board when the processor crashed. All these signals were available on headers on the board, and cable connectors were used to connect wires to them. The board has a power plug which must be supplied with a DC voltage of about 8 V or higher for the on-board regulator to supply 5 V. Some of the connectors can be seen in Figure 4.8.

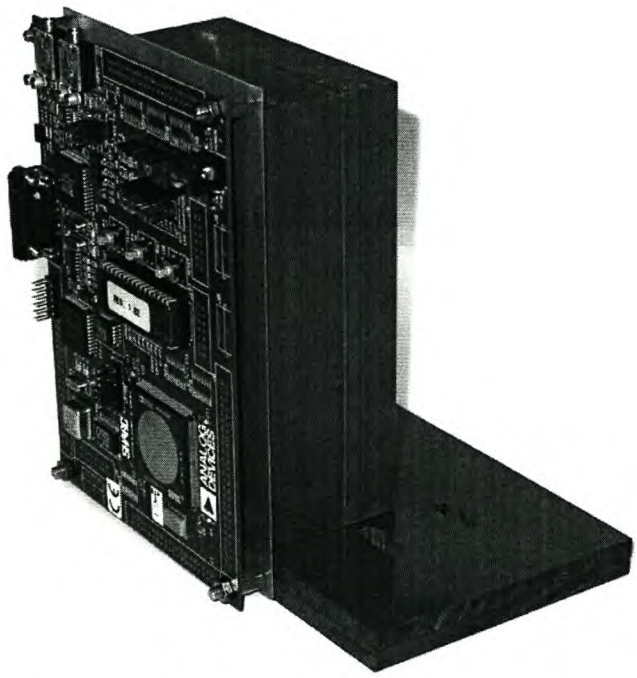


**Figure 4.8:** A diagonal view of the evaluation board mounted on the brass block with a tufnel base.

The ADSP-21061 DSP processor irradiated was fabricated with a 0,45 $\mu$m embedded SRAM CMOS process. The size of the processor die is 10,34 $\times$ 10,54 mm and the package dimensions are 32 $\times$ 32 mm. The package has a built-in heat slug which has a diameter of 24,1 mm and is 1,37 mm thick. The package was made from Ortho-Creosol-Novolac (OCN) type resin.

### Connections Between Patch Board and Evaluation Board

The signals that were sent to, and received from the computer room were the following:

- Temperature output (to computer)
- Power switch (from computer)
- Board reset (from computer)
- Latchup interrupt (to computer)
- 3× Flag bits (to computer)
- Serial port (to and from computer)

A basic description of these signals was given in Section 4.4.1. The temperature signal was not used for the tests in the therapy station because temperature measurements in free air could be done anywhere. For the tests performed at the vacuum tank, the signals passing to and from the inside of the tank were:

- +12 V power
- 3x Flag bit signals of DSP processor
- Serial port
- Evaluation board reset
- Temperature measurement

All these signals, except the 12 V power, are connected to the patch board via the control board.

### Radiation Room Control Board

The control board used in the radiation rooms connects the computer control board to the evaluation board and has other purposes as well. It switches the power to the evaluation board on and off and measures the current drawn by the evaluation board. When a latchup occurs it turns the power off and lets the computer know. The control board also buffers the digital, analogue and RS232 signals transmitted to, and received from, the computer room.

How these functions were implemented and their operation are discussed below. A schematic of each function together with a brief description is given in each case.

During irradiation of the DSP processor latchup can occur, which can destroy the processor if the power is not switched off. (Refer to Section 3.2 for more information on single event latchups). To turn the power off, a switch was implemented with two MOSFETs. The configuration uses a n-channel and p-channel MOSFET as seen in Figure 4.9. A MOSFET was used as a switch instead of a bipolar transistor because of the low resistance when switched on ($R_{DS(on)} = 0,20\ \Omega$ for the IRF9530N), which implies a low source-drain voltage drop across the MOSFET. The VN10K MOSFET is used to switch the IRF9530N on and off. The power switch is controlled by the "power switch" and "latchup control" signals. The two signals are AND'ed together with diode logic, and then buffered by two NOT gates before driving the VN10K MOSFET.

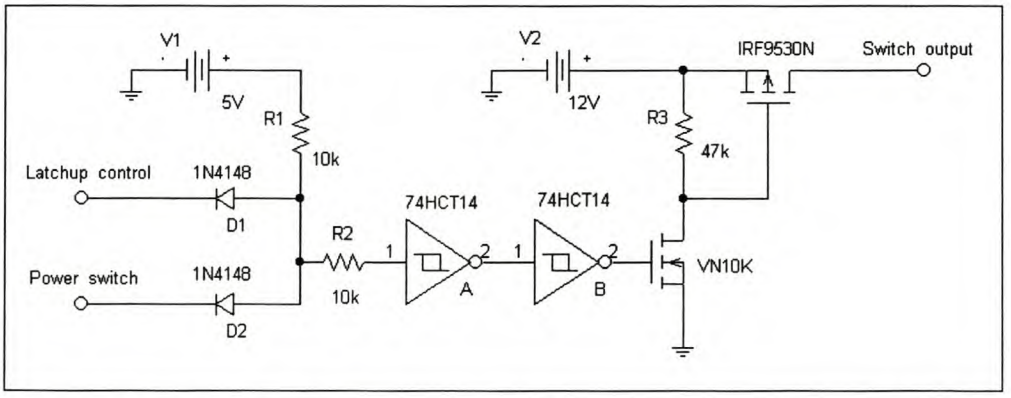



**Figure 4.9:** A switch to turn power to the evaluation board on and off. The switch is controlled by the combination of the "power switch" and "latchup control" signals. The diode logic implements a logical AND gate.

A current meter was implemented after the switch in the circuit to be able to measure when a SEL occurs. This was done with a differential op-amp, which measures the voltage across a resistor. The resistor is placed in series with the output of the switch, so all current to the evaluation board passes through it. See Figure 4.10 for the schematic. A few half-watt resistors were put in parallel to obtain a $0,46\ \Omega$ resistor. The small resistor ensures that the voltage drop across it is small. The differential amplifier has a gain of $100k/18k = 5,55$. When the evaluation board draws a typical current of 500 mA, an output of 500 mA $\times$ 0,46 $\Omega \times 5,55 = 1,27$ V is given by the current meter.

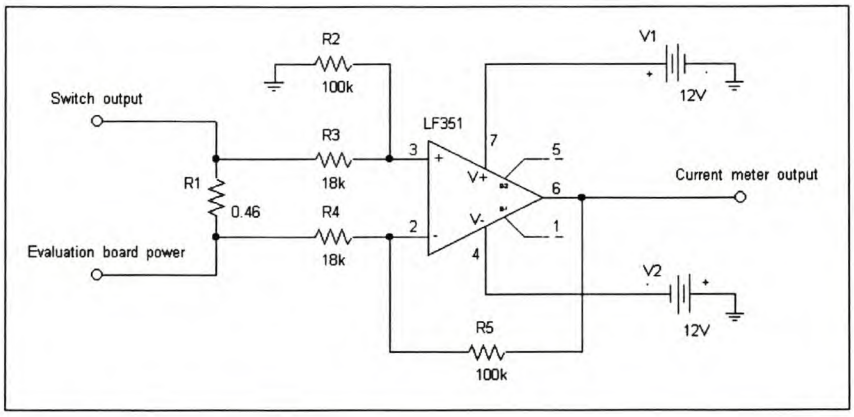



**Figure 4.10:** A circuit for measuring the current drawn by the evaluation board.

When latchup occurs in the processor, a higher than normal current is drawn. An op-amp comparator is used to test for this condition by comparing the output of the current meter with a preset voltage, which is set by a pot. The output of the comparator is high while the current measurement voltage is lower than the reference one. As soon as the measurement voltage passes the reference voltage, the output goes low. This is used to trigger a 555 timer, which creates a delay. See Figure 4.11 for the schematic.

The timer is configured for monostable operation, and its output goes high for a time of 113,2 ms when triggered. The output is inverted with a NOT gate, which turns the power switch off when the timer is triggered. The output of the timer also sends an interrupt to the computer via the parallel port. This is to let the computer know that a latchup has occurred. The timer goes high long enough for the computer to receive and react on the interrupt, which is to make the "power switch" signal low. This ensures that when the timer output goes low again, the "power switch" signal will keep the switch off until the computer turns it on.

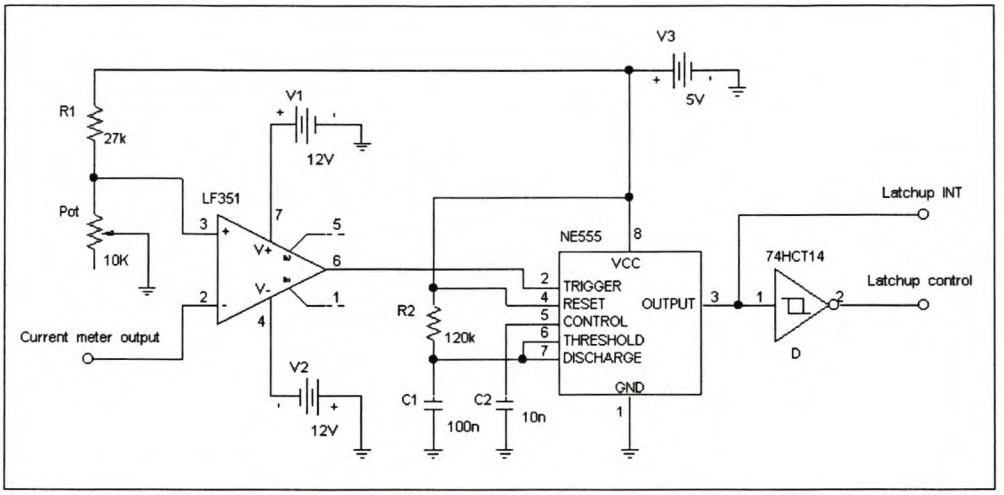



**Figure 4.11:** The latchup protection circuit with a delay to make sure that the computer has time to receive and react on the interrupt.

The DSP processor has an in-package copper heat sink to improve power dissipation. A LM335 temperature sensor was used to determine how much heat the processor generates in a vacuum. The LM335 outputs a voltage of 10mV/°K. This voltage is buffered by an op-amp with a gain of 1,01 before going to the computer. A 10 Ω resistor is used in series with the op-amp output to keep it stable when driving the coaxial cable. See Figure 4.12 for the schematic. An analogue sampling card was used in the computer to digitise the temperature voltage. The temperature cables were not used for the tests in the therapy station because the processor was in free air there.

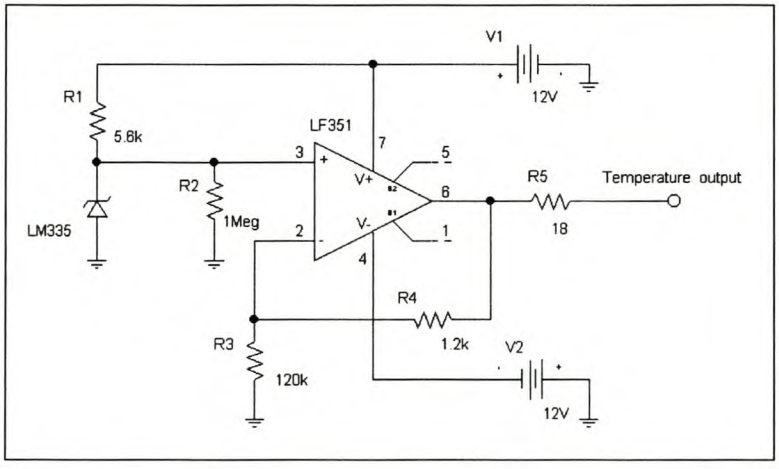



**Figure 4.12:** The circuit used to measure the temperature of the DSP processor.

The digital signals were all buffered to ensure that the signals were not distorted by the coaxial cable and to prevent loading the evaluation board circuits. The buffer is described in detail in Section 4.4.1 and the circuit can be seen in Figure 4.4. The serial port RS232 signals were buffered in a similar way as the digital ones, except for the fact that the RS232 signals swing positive and negative. A push-pull amplifier was used to solve this, as can be seen in Figure 4.5 in Section 4.4.1.

An open-collector transistor configuration was used to reset the evaluation board with the "board reset" control signal. This had to be used because there is another open-drain signal driving the same reset pin. The open-drain circuit is shown in Figure 4.13.

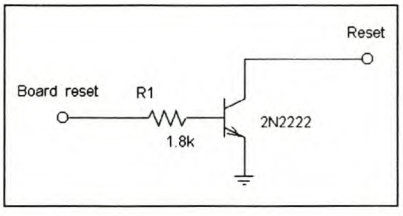



**Figure 4.13:** The open-collector circuit used to reset the evaluation board.

A 7805 voltage regulator was used to supply 5 V to all the logic circuits from the 12 V used for the serial buffer.

## 4.5 Test Software

This section explains the interface program run on the computer and the four test programs run on the combination of the computer and DSP processor. The function of the test programs and the code used to implement them are covered in each case.

### 4.5.1 Programs Used for Test

The software used on the computer in the computer room was a Pascal program and a download program. The Pascal program served as the user interface where the user can

choose between the four different tests. The download program came with the evaluation kit and allows downloading and running programs on the DSP processor. The file to download and what to do afterwards is specified with a command-line interface. The Pascal program used it to start each test on the processor. The Pascal program communicated with the DSP processor through the serial port. This is done via a serial host interface which loads from an EPROM on the evaluation board each time it boots. The download program makes use of this program to download programs to the processor as well.

The test program on the computer ran in DOS mode to ensure that it kept up with the data received from the evaluation board. Pascal was chosen as programming language because it was faster to program in, and the routines to read the sampling card and handle serial communications were available.

The computer parallel port was used for controlling the control circuit board and the evaluation board in the vacuum room. Signals are read back from these boards through the parallel board as well. The parallel port interrupt pin was used by the vacuum room control board to let the computer know that a latchup has occurred on the DSP processor. This triggers an interrupt, which makes the "power switch" signal low to switched off the power to the evaluation board.

An Analogue to Digital (A/D) converter on a sampling card in the computer was used to read the voltage returned from the vacuum tank temperature sensor. The card has a few Input/Output (I/O) ports which are used to start sampling and read back the sampled values. See Appendix F for the code used. The temperature measurement is stored at the start and end of each test run on the DSP processor.

Throughout the test all operations performed and important data received is echoed to the screen and written to a report file. The report file is a plain text file which contains similar messages to those output to the screen. Each message is written together with the current time so the passage of the test can be followed afterwards.

The NOP, Cache and ALU tests use similar startup code for each procedure. This entails the following operations:

- Open a report file to store the progress of the test
- Test if power is off due to a latchup
- Turn power on
- Reset evaluation board
- Measure DSP processor temperature
- Download the test program to DSP processor

The test procedures function in different ways after this. The code for the NOP, Cache, ALU and Memory test programs are discussed below. Each test was implemented as a procedure in the Pascal program. The NOP, Cache and ALU tests also have programs that run on the DSP processor, which are discussed as well.

### 4.5.2 NOP Test Software

The NOP test determines the susceptibility of the DSP processor's program counter to upsets. It tests this by running through four blocks of code and jumping crosswise from one to the other. At the start of each block, flags are set to indicate in which block it currently is. The computer program monitors the flag values to check if the program runs through the blocks in the right order. If the order is broken, an upset is logged.

The details of the NOP program running on the DSP processor and the test procedure on the computer are discussed below.

#### NOP Program on DSP Processor

The three flag pins Flag 0, Flag 2 and Flag 3 are used for the test. Flag 1 is wired as an input pin on the evaluation board, and cannot be used. At the start of the NOP program the three flag pins are set up as output pins. Flag 0 is used as clock, while Flags 2 and 3 indicate the current block for the test.

When the program runs, it jumps from one block of NOP operations to another. At the start of each block, it sets Flags 2 and 3 to represent a binary number, and toggles Flag 0 to indicate the flags have changed. The blocks are padded with 760 NOP operations to create a delay between setting the flags and jumping to the next block. This gives the computer enough time to read the flags and check the course of the program. The program runs and sets the flags in the following order: The first block sets the number to 0 and jumps to the third block. In the third block the number is set to 1 and jumps to block 2. Block 2 sets the number to 2 and jumps to the fourth block. There it sets the number to 3 and jumps to the start (block 0). See Appendix E for a flow diagram and shortened code of the NOP program running on the DSP processor.

### NOP Test Procedure on Computer

The NOP procedure starts with the operations in the above list. After the last item on the list has finished, the program runs in a loop. In the loop it continuously reads the flag bits from the parallel port. The Flag 0 bit, which is used as a clock, is monitored for a high to low or low to high transition. When it changes, the other two flag bits are compared to the number of the next block. If they do not match, the error is logged as an upset, and the program continues with the upset number, otherwise the number of tests is incremented and it continues the loop.

The program has a timeout check for the flag bits so that if they do not change in a certain time the program assumes that the program has crashed. It then displays the amount of upsets and total tests run since it started. After the timeout the test restarts from the point: "Test if power is off due to a latchup" in the above list. During the test the user can end testing by pressing "ESC", after which a summary is displayed and the Pascal program returns to the main menu.

See Appendix F for a flow diagram and the code of the NOP test procedure.

### 4.5.3 Cache Test Software

The Cache test determines the susceptibility of the DSP processor's cache and program counter to upsets. It works the same way as the NOP test, except that the jump instruction at the end of each block is cached, so that upsets in the cache will make the program jump to wrong locations. This is picked up by the Pascal program, which checks the order of the numbers returned by the flag bits.

#### Operation of ADSP-21061 Instruction Cache

The ADSP-21061's on-chip instruction cache is a 2-way, set-associative cache with entries for 32 instructions. The operation of the cache is transparent to the programmer. The ADSP-21061 caches only instructions that conflict with Program Memory (PM) data accesses over the PM Data Bus. This feature makes the cache considerably more efficient than a cache that loads every instruction, since typically only a few instructions must access data from a block of PM.

If an instruction at address $n$ requires a PM data access, there is a conflict with the instruction fetch at address $n+2$, due to the three-stage instruction pipeline. This causes the instruction at $n+2$ to be stored in the instruction cache. If the instruction needed is in the cache, a "cache hit" occurs and the cache provides the instruction while the PM data access is performed. If the instruction needed is not in the cache, a "cache miss" occurs, and an instruction fetch from memory takes place in the cycle following the PM data access. This incurs one cycle of overhead, and the instruction is loaded into the cache so that it is available the next time the same instruction is executed.

The cache contains 32 entries. An entry consists of a register pair containing an instruction and its address. Each entry has a "valid" bit which is set if the entry contains a valid instruction. The entries are divided into 16 sets, numbered 15–0, of two entries each, entry 0 and entry 1. Each set has a Least Recently Used (LRU) bit that indicates which of the two entries contains the least recently used instruction.

Every possible instruction address is mapped to a set in the cache by its 4 Least Significant Bits (LSBs). When the processor needs to fetch an instruction from the cache, it uses the 4 address LSBs as an index to a particular set. Within that set, it checks the addresses of the two entries to see whether either contains the desired instruction. A cache hit occurs if the instruction is found, and the LRU bit is updated to indicate the other entry. A cache miss occurs if neither entry in the set contains the needed instruction. In this case a new instruction and its address are loaded into the least recently used entry of the set that matches the 4 LSBs of the address. The LRU bit is toggled to indicate that the other entry in the set is now the least recently used. [1]

#### Cache Program on DSP Processor

The Cache program uses the same code as that of the NOP test, except for the addition of two instructions to cause the jumps at the end of each block to be cached. The jump instruction is "forced" into the cache by executing a dummy PM read and NOP operation before the jump. See above for an explanation of the cache operation. To prevent the jump instructions from overwriting each other in the cache, the lower four bits of no more than two of the instruction addresses may be the same. See Appendix E for a flow diagram of the program and a shortened code listing.

#### Cache Test Procedure on Computer

The Pascal code for the Cache test is the same as that of the NOP test. A flow diagram of the procedure and the code listing can be seen in Appendix F.

### 4.5.4 ALU Test Software

The ALU programs test for upsets in the DSP processor's Arithmetic and Logic Unit (ALU) during irradiation. This is done by the processor doing various logic and arithmetic calculations, and then testing if the result is correct. If any upsets occurred during the calculations, the result will show it. The Pascal program counts the number of upsets and tests, and logs the results.

### ALU Program on DSP Processor

At the start of each test loop, the start and end values for the logical and arithmetic tests are checked against two other identical numbers. If they differ the program tries to correct the values. If unsuccessful it stays in a small loop to create a timeout in the Pascal procedure, which will restart the test.

The testing of the ALU is done with a program running on the DSP processor which does a series of logic operations like shift left, shift right, AND, OR, NOT and XOR. The logical computations use the same 32-bit start value each time. After the operations are done, the result is compared with the stored result. If they differ Flag 2 is set, otherwise it is cleared. A series of arithmetic operations covering $+, -, \times$ and $\div$ are performed after the logical test. For each test the same 32-bit start values are used, and the result is compared with the correct one. Flag 3 is set for an error in the result and cleared otherwise. When the two tests are done, Flag 0 is toggled to signal that both tests have finished. A short delay is created by a loop to give the computer time to read the flags. See Appendix E for a flow diagram and code listing of the program.

### ALU Test Procedure on Computer

The Pascal program runs in a loop waiting for Flag 0 to change state. As soon as it changes, it adds the Flag 2 and 3 states directly to counters, since the flags are set when upsets occur. The total number of tests, as well as the number of logical and arithmetic upsets are stored in separate counters. At the end of the ALU test the counters are written to the report file.

A timeout check is used to restart the test when Flag 0 does not change within a certain time. When this happens the procedure displays a summary of the test, which includes the total number of upsets and tests run since it started. The test restarts from the point: "Test if power is off due to a latchup" after a timeout. During the test the test can be stopped by pressing "ESC". The Pascal program displays a summary of the test and returns to the main menu when "ESC" is pressed.

### 4.5.5 Memory Test Software

The function of the memory test is to determine how many upsets can be expected in the on-chip memory of the DSP processor. In general memory tends to be more susceptible to upsets than other components. Memory upsets can lead to random processor behaviour, due to program and/or data memory being corrupted.

The memory test program makes use of the serial host interface described below. It is used to write a checkerboard pattern to the processor memory, and then check for any changes. This is all done on the computer, which reads and writes blocks of memory with the host interface. Any memory upsets are logged and the bytes corrected on the processor.

A brief description of the serial host interface is given below, followed by the Pascal memory test procedure.

#### EZ-Kit Lite Serial Host Interface

The EZ-KIT Lite comes with a monitor program which is loaded when the processor boots from an onboard EPROM. The monitor program provides a host interface to the DSP processor through a RS-232 serial port. By sending and receiving message packets, the serial host interface allows one to perform the following operations :

- Write block of data memory
- Write block of program memory
- Read block of data memory
- Read block of program memory
- Read core registers
- Start executing code on processor
- Reset the processor
- Reset the board
- Set board serial port speed

- Verify communications with host
- Resynchronise communications with host

These commands are executed by sending a message packet to the DSP processor. The packet header contains the source ID, destination ID, packet ID and packet length. Any extra information and data follows the header. The processor responds with a similar packet to signal that the command has been done. The information requested follows the packet header, or if the command was unsuccessful an error code is returned.

A procedure "Get_packet" was written to check packets for errors, and display information on the packet received. The information and errors are written to the report file as well. "Get_packet" removes the packet header and leaves the data in the serial buffer for the test program to process. See Appendix F for the code listing of "Get_packet".

To make sending a message packet easier, all the message packets were declared as constant arrays. It is simpler to understand the program when the arrays are sent as pointer parameters to the "Send_serial_data" procedure. The "Send_serial_data" procedure sends all the bytes at the pointer address to the serial port. The first byte the pointer points to is the number of bytes that "Send_serial_data" has to send, and the rest are the actual bytes. If certain parameters in the packet are determined by the test procedure itself, they have to be sent manually afterwards with "Send_serial_byte".

### Memory Test Procedure on Computer

The ADSP-21061 has a total of 1 Mbit SRAM on-chip, which is split up into 384 kbit Data Memory (DM) and 640 kbit Program Memory (PM) for the test. The DM is 32 bits wide, and the PM is 48 bits wide. Not all the memory can be tested since the host interface uses some program and data memory. The memory tested is PM addresses in the range 20300 hex – 21FFB hex and DM addresses in the range 23000 hex – 27E70 hex. This gives a total of 975 kbits that were tested.

The memory test makes use of the serial host interface to read and write memory blocks, reset the processor and configure the serial port of the evaluation board. The serial host

interface is described above. At the start of the memory test the following operations are performed:

- Open a report file to store the progress of the test
- Test if power is off due to a latchup
- Turn power on
- Reset evaluation board
- Reset DSP processor
- Set serial speed to 115200 baud
- Verify communications

The rest of the test involves writing 55 hex (01010101 binary) to all available data and program memory locations. It reads and stores the core register values as well. The test runs in a loop which reads and compares the memory with 55 hex to see if any upsets occurred. Any changes are logged, and corrected by writing 55 hex to that address again. The DM is checked first, then the PM and the core registers last. The test compares the core register values with their previously stored values to detect upsets. If the values change, an upset is logged and the upset values are stored. The program loops back to the DM test hereafter.

During the test the host interface program can be upset at any time and cause wrong data to be returned, or nothing at all. To prevent this from crashing the Pascal program, the "Get_packet" procedure checks for corrupt packets, and timeouts determine if the program on the processor is still running. The timeout checks restart the test from the point where the program tests if a latchup has occurred if there is no response.

At the end of the test (when "ESC" is pressed) the program writes the total number of blocks of memory tested and the number of upsets to the report file. See Appendix F for a flow diagram and code listing of the procedure.

## 4.6 Procedure of Testing

The tests consisted of getting everything set up and and doing the tests. The following text explains these procedures for the two test locations that were used. The staff at the NAC helped with the tests. Dr Kobus Lawrie organised the tests and helped with the calculations at the vacuum room. When the tests did not work out he arranged that Mr Evan de Kock took over the tests at the proton therapy station. At the therapy station Mr J.E. Symons and Dr D. Nichiporov provided assistance during the tests as well.

### 4.6.1 Preparing hardware setup

There were two locations where the SEU tests were performed. The preparation of the hardware setups for both are described below.

#### Test in Vacuum Tank Room

The first priority at the start of the tests in the vacuum room was to get the setup finished so all the connections could be tested before the vacuum tank was closed. The setup comprises the computer and board in the data-logging room and the control board in the vacuum room with the evaluation board inside the tank. The irradiation can only start once the pressure in the tank is low enough for the connecting valves to open up, unblocking the beam. The vacuum tank took about two hours to reach this pressure after it was closed.

Once the tank was ready for irradiation the beam had to be steered to the tank with the steering magnets. A ruby inside the tank is used to align the beam with the centre. This is done with a camera viewing the ruby, which glows when irradiated. The tests could start when this was all finished.

**Test in Proton Therapy Station**

For the radiation therapy tests the preparation started with setting up the computer and the control board in the control room. This involved supplying power to the control board and connecting it to the computer and patch board.

As soon as the therapy station was available, the chair used for patients was lowered beneath the floor level and a trolley with a water tank on top rolled into its place. The tank is used for beam calibration. The beam was then calibrated for the energy we wished to use with a test run. Hereafter the test hardware was set up and connected in the room. The brass block collimator with the evaluation board on it was placed on the water tank trolley. Spacers were used to get it to the correct height, and the hole in the brass was aligned with the help of a laser beam. The control board was supplied with power and connected to the patch board and evaluation board.

Once everything was set up the test programs were run to make sure that all the connections were correct and working. Everything was ready to start testing hereafter.

### 4.6.2 Performing SEU Testing

The process of testing started with a specific test program being started on the computer. In most cases the tests were done in the order NOP, cache, ALU and then memory test. Once the program had been downloaded to the DSP processor and was running properly, the irradiation could be started. The flux rate was set to the desired value and then the irradiation was started. This continued until the total fluence of $10^{10}$ p/cm$^2$ was reached or enough upsets had been recorded. The irradiation was stopped when the host stopped responding during the memory test program or the download program stopped at the start of the other three tests. In the case of the memory test the processor had to be reset and the value 55 hex written to the memory before the test could continue. When the download program stopped the test program on the computer had to be restarted before the irradiation could resume.

The fluence at the vacuum chamber tests was recorded with counters which integrate the dose rate. At the start of each test the counters had to be reset and at the end the fluence measurement written down. The beam had to be started and stopped by hand each time. The flux rate was approximately $3 \times 10^9$ p/cm$^2$/s which left little time to run the tests. This was a limitation of the measurement setup which could not measure low flux rates.

The proton beam at the proton therapy station was controlled by a computer in the physics laboratory. The total desired dose was typed in and when that dose was reached the beam was switched off automatically. During irradiation the beam could be stopped at any time and switched on again. The computer gave the total fluence reading during the test and continued with the counter if the irradiation was continued after a stop. This reading was used in the cases where the test was stopped before a fluence of $10^{10}$ p/cm$^2$ was reached. The flux rates were chosen as $8,1 \times 10^7$ p/cm$^2$/s or lower, which gave a test time of at least 123 seconds. This was enough time to record sufficient data.

The name of the report file together with the flux rate and total fluence received were written down after each test was finished. The flux rate for the next test was then set and the test program started.

## 4.7 SEU Test Results

After all the tests were done the results were processed. This entails counting the number and type of upsets for each test. The number of errors for each test are tabled below. The device error probability or cross-section is then calculated for all the tests to generate a graph of proton energy against the device cross-section. This graph can be used to calculate typical upset rates in space.

The cross-section, $\sigma$, of a test is calculated by:

$$\sigma = (N/F) \sec \theta$$

where N is the number of errors and F is the beam fluence. The angle $\theta$ is the incident angle of the beam measured with respect to the chip surface normal. [22]

The following subsections discuss the results and conclusions from the tests performed at the two locations. The cross-sections are compared with ones of SEU tests previously performed.

### 4.7.1 Tests in Vacuum Room

The tests that were performed at the vacuum chamber did not provide any proper results. The measurements that can be given are that a total fluence of $7,23 \times 10^{11}$ p/cm$^2$ was reached and latchup did occur in this irradiation period. The accuracy of the measurements were questionable with the measurement method used, so a higher fluence is possible. With the setup used there was a possibility that neutrons could be present in the beam. Some of the internal memory bits sustained permanent damage during the experiments, which could be as a result of neutrons. The permanent memory damage caused concern for the useability of the memory in a space environment. The tests in the therapy station did not cause any latchups or permanent damage after a total fluence of $1,20 \times 10^{11}$ p/cm$^2$ however. This indicated that the test setup was the cause of the latchups and memory damage for the first tests, and that the memory was not as sensitive as feared.

### 4.7.2 Tests in Proton Therapy Station

The first SEU test was performed with 200 MeV protons. For the second test runs 65 and 45 MeV protons were used. The beam fluxes and fluences used for each test are listed in Table 4.3. The upset results are listed in Tables 4.4 to 4.7.

**Table 4.3:** The parameters of the proton beams that were used for the different SEU tests.

| Test | Energy (MeV) | Flux ($p/cm^2/s$) | Fluence ($p/cm^2$) |
|---|---|---|---|
| NOP | 200 | $8,1 \times 10^7$ | $1 \times 10^{10}$ |
| Cache | | $8,1 \times 10^7$ | $8,65 \times 10^9$ |
| ALU | | $2,6 \times 10^7$ | $1,03 \times 10^{10}$ |
| Memory | | $2,6 \times 10^7$ | $8,9 \times 10^9$ |
| NOP | 65 | $8,1 \times 10^7$ | $1,09 \times 10^{10}$ |
| Cache | | $8,1 \times 10^7$ | $1,09 \times 10^{10}$ |
| ALU | | $2,6 \times 10^7$ | $1,09 \times 10^{10}$ |
| Memory | | $2,6 \times 10^7$ | $9,92 \times 10^9$ |
| NOP | 45 | $8,1 \times 10^7$ | $1 \times 10^{10}$ |
| Cache | | $8,1 \times 10^7$ | $1 \times 10^{10}$ |
| ALU | | $8,1 \times 10^7$ | $1 \times 10^{10}$ |
| Memory | | $2,6 \times 10^7$ | $1 \times 10^{10}$ |

**Table 4.4:** The type and number of upsets for different proton energies during the NOP test.

| | Proton Energy (MeV) | | |
|---|---|---|---|
| **Type of Error** | 200 | 65 | 45 |
| NOP error | 0 | 0 | 0 |
| Stopped responding | 2 | 1 | 0 |

**Table 4.5:** The type and number of upsets for different proton energies during the Cache test.

| | Proton Energy (MeV) | | |
|---|---|---|---|
| **Type of Error** | 200 | 65 | 45 |
| Cache error | 0 | 0 | 0 |
| Stopped responding | 3 | 1 | 2 |

**Table 4.6:** The type and number of upsets for different proton energies during the ALU test.

| | Proton Energy (MeV) | | |
|---|---|---|---|
| **Type of Error** | 200 | 65 | 45 |
| ALU error | 5 | 2 | 6 |
| Stopped responding | 0 | 1 | 0 |

**Table 4.7:** The type and number of upsets for different proton energies during the Memory test.

| | Proton Energy (MeV) | | |
|---|---|---|---|
| **Type of Error** | 200 | 65 | 45 |
| Memory Bit Upset | 512 | 527 | 460 |
| Register Bit Upset | 0 | 0 | 0 |
| Stopped responding | 4 | 6 | 4 |

All tests were performed with the beam perpendicular to the surface of the chip. This results in the $\sec\theta$ term of the device cross-section calculation being equal to 1. The graphs of the proton energy against device cross-section are given in Figure 4.14.

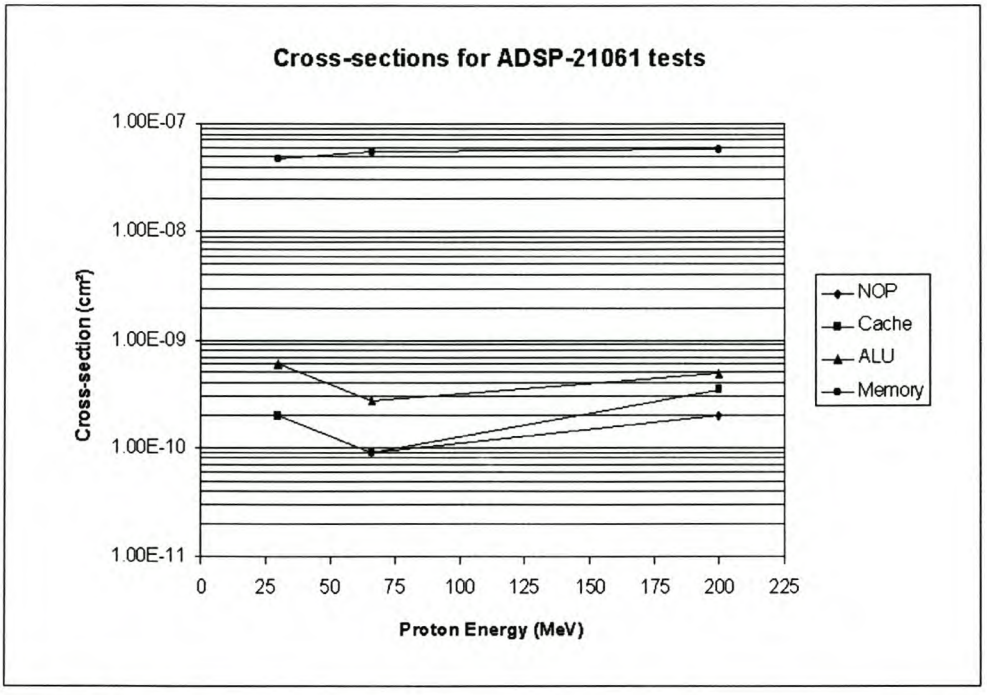



**Figure 4.14:** A graph of proton energy against the device cross-section for the four tests

No latchups were experienced during testing which is very good. Devices that do latchup are usually not considered for space use. From the cross-section graph (Figure 4.14) it can be seen that the NOP and Cache tests are very similar, which indicates that the use of the cache does not severely affect the processor. The on-chip memory is two orders more sensitive to upsets than the rest of the processor. This means that calculations will have to be done to determine if this sensitivity is acceptable. If the internal SRAM is used, an EDAC circuit should be added to prevent corruption of it.

The cross-section results can be used in conjunction with special software to calculate typical upset rates for a certain orbit. Some of the programs and models that provide this function are CREME, AE-8MIN, AE-8MAX, AP-8MIN, AP-8MAX, Radbelt, Solpro and SHIELDOSE. This software was not available so the SEU results were compared with previous tests. The cross-section graphs of three processors are given below in Figures 4.15 to 4.17. They are for the TMS320C25 DSP, Pentium MMX and Pentium II processors.

To get a rough estimate of the DSP processor upset rate, the flux variations in Figure 3.4 were used to calculate the average flux for an orbit through the SAA. A proton flux of 1 proton/cm$^2$/s was used for the rest of the orbit around the earth. This results in an average of about 60 protons/cm$^2$/s for the whole orbit. The average flux was used together with the average cross section of $5.5 \times 10^{-8}$ cm$^2$ for the memory test to determine an upset rate. SUNSAT took about 100 minutes to complete one orbit. This results in an upset figure of 0,0198 upsets/orbit.

One aspect of the test results which does not follow the same tendency as previous tests is the cross-sections for the 45 MeV cache and ALU tests. Previous tests had a curve which falls fast at the lower proton energies. A possible explanation is that the processor did not have time to recover from the test run because the 65 and 45 MeV tests were done directly after each other. The NOP and memory tests produced the expected results however (the NOP test had no upsets), which does not support this theory. The other option is that it is due to statistical variance. Unfortunately there was not enough time to repeat these two tests to confirm the results.

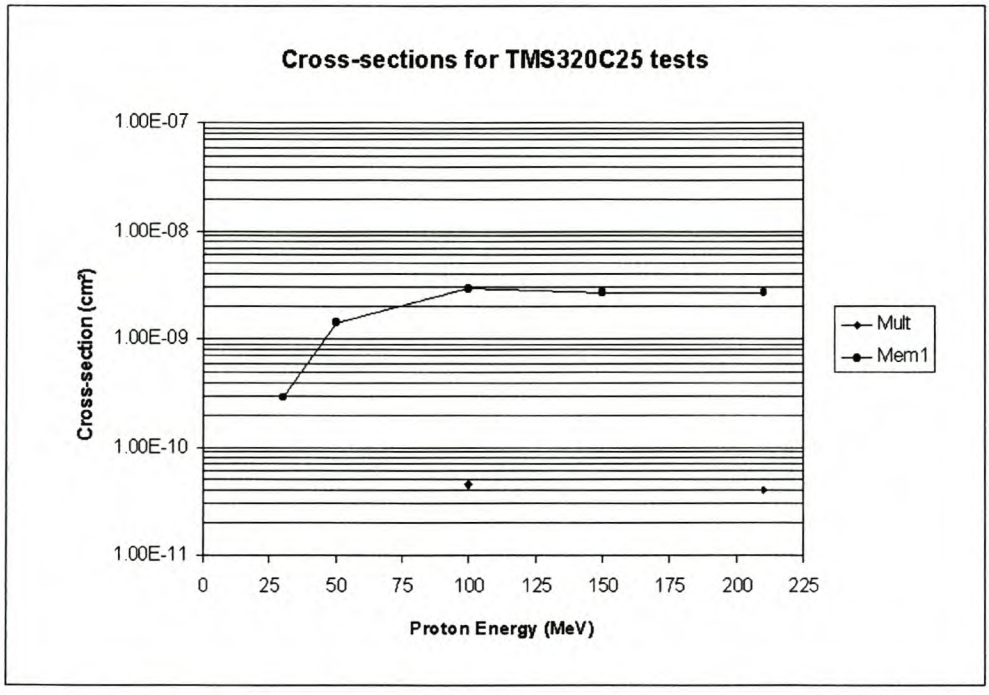



**Figure 4.15:** The cross-section results of previous testing of the TMS320C25 DSP processor. [29]

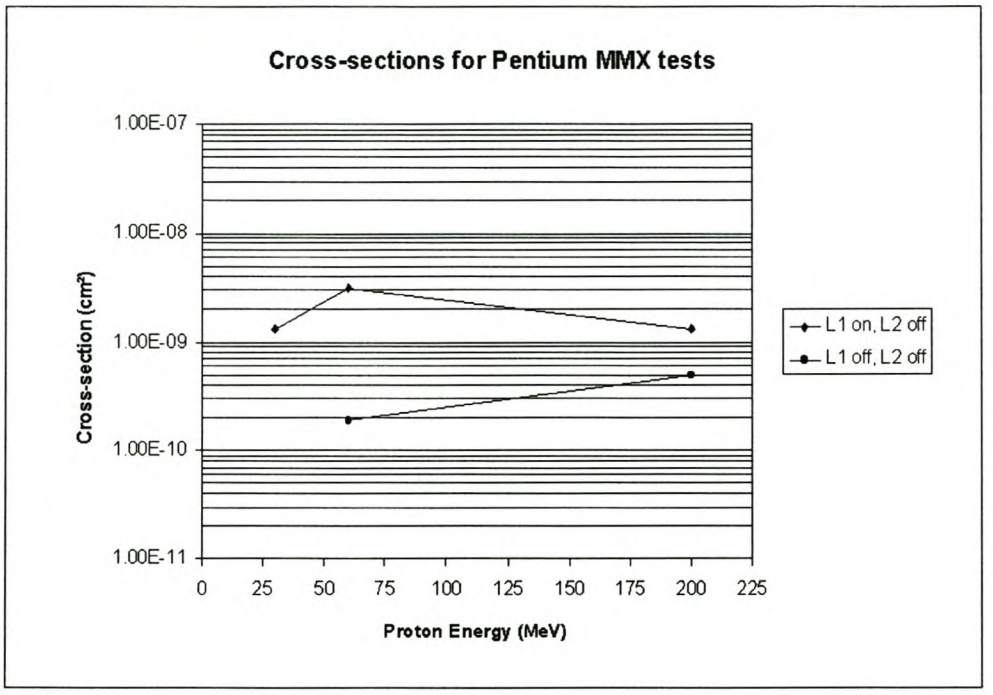



**Figure 4.16:** The Pentium MMX cross-section results from previous SEU testing. [15]

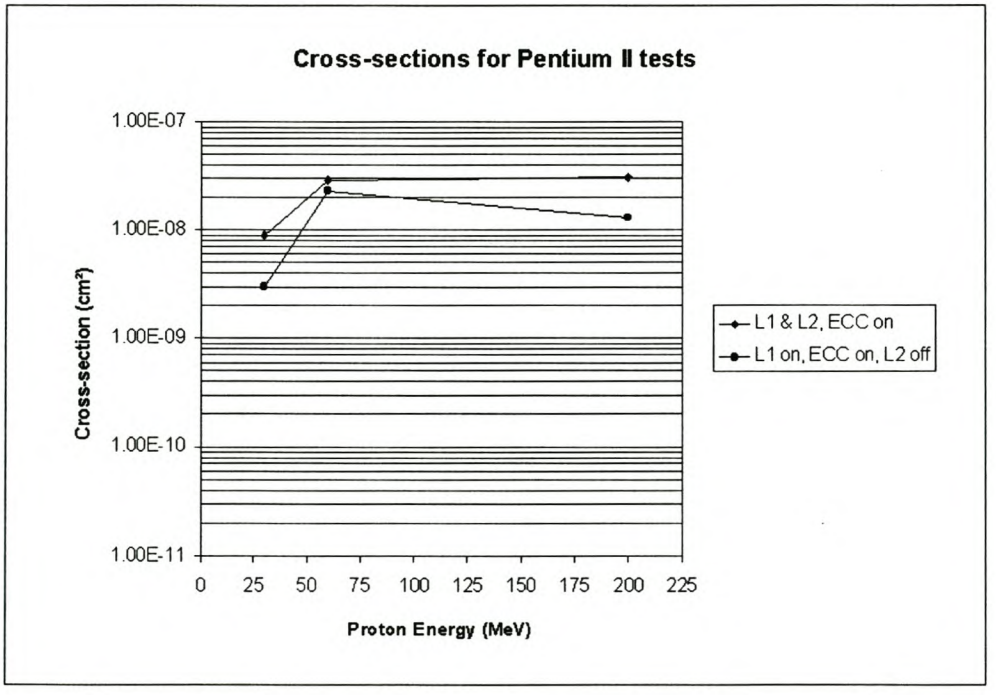



**Figure 4.17:** A graph of proton energy against device cross-section for the Pentium II processor. [15]

## 4.8 Recommendations for future tests

After the tests were performed at two different locations and the test setups were thoroughly tested, a few recommendations can be made for future SEU testing. This will help to make tests easier and faster next time.

The proton therapy section is definitely the best location for SEU testing. The setup can be assembled in a short time whereafter testing can begin. No extra time is required to generate a vacuum. If there is a problem during the test the beam can be switched off and one can check the setup in the proton therapy station. The measurements were more accurate as well.

The method where the parallel port was used to count the number of errors can be improved by using a microprocessor instead. A microprocessor can read the flag pins much faster than the parallel port of the computer can, which allows the tests on the DSP processor to run faster. Another part which can be improved is the downloading of the test programs to the DSP processor. The serial port is not very fast, which meant there was a few seconds delay each time the software had to be reloaded. A better solution would be to connect an extra EPROM to the board to store the test programs. The computer can then send a command to load and execute one of the programs.

The lowest proton energy used was 45 MeV. Lower energies can be used for testing if different measuring equipment is used. This has to be organised and tested before the tests are done. One should try to use these lower energy protons in future tests.

# Chapter 5

# Implementing an EDAC Circuit for the ADSP-21061

An Error Detection And Correction (EDAC) circuit was developed to detect and correct errors in the internal memory of the ADSP-21061. This is so that the internal memory can be used with confidence in space. Special hardware had to be designed to check the internal memory. Any external memory added to the processor can easily be checked with commercial EDAC circuits. One of the design parameters for the EDAC circuit was that the overhead for the DSP processor when using the EDAC circuit must be kept to a minimum. Various configurations for the circuit were considered, and are discussed below.

## 5.1 Reasons for Using an EDAC Code

An Error Detection And Correction (EDAC) code allows one to check any form of data for corruption and to correct the data. The number of corrupted bits that can be detected and corrected depends on the EDAC code used. Various EDAC codes have been developed for specific applications and each has advantages and disadvantages.

EDAC codes are used to make data transmission faster and more robust. This is achieved in spite of the extra bits needed for detection and correction because whole data blocks

have to be retransmitted when no error correction is used and data is corrupted. Another application where EDAC codes are used is in circuits which check and correct upsets in memory. These circuits are used particularly for processor systems in space. It is very important that processor instructions stay uncorrupted, because they can perform random operations due to an upset, which can have fatal consequences.

## 5.2 EDAC Codes

EDAC codes use extra bits to enable the detection and correction of data errors. These extra bits have to be transmitted or stored together with the data bits. Different codes have different ways of computing the extra bits and checking for errors afterwards. The more check bits are used, the more corrupted bits can be detected and corrected. If more errors occur in a block of data than the code can detect, the result of the code is not valid anymore. This can result in the wrong bits being corrected and the data becoming even more corrupt. A compromise between the amount of extra bandwidth or storage space used, and the ability to detect and correct errors has to be made. This is mainly influenced by the amount of upsets expected in the data for a specific noise level or radiation environment.

To explain how EDAC codes work, take a frame of $m$ data (message) bits and $r$ redundant or check bits. The total length $n = m + r$ is often referred to as a $n$-bit codeword. The number of bit positions in which two codewords differ is called the Hamming distance. Its significance is that if two codewords are a Hamming distance $d$ apart, it will require $d$ single-bit errors to convert one into the other. The error detecting and correcting capabilities of a code depends on its Hamming distance. To detect $d$ errors, you need a distance $d + 1$ code because with such a code there is no way that $d$ single-bit errors can change a valid code-word into another valid codeword. Similarly to correct $d$ errors, you need a distance $2d + 1$ code because that way the legal codewords are so far apart that even with $d$ changes, the original codeword is still closer than any other code-word, so it can be uniquely determined. This allows correction anywhere in the codeword, including the check bits. [11]

There are many EDAC codes available, each with its own properties. Many of the codes are intended for data communications. A feature of these codes is the ability to handle burst errors, where a whole string of bits can be corrupted. Two EDAC codes were considered for checking the ADSP-21061 memory, the Hamming and Bose-Chaudhary-Hoquenghem (BCH) codes.

Most forms of the Hamming code correct 1-bit errors, and detect 2-bit errors. The basic Hamming code uses 5 check bits for 8 data bits. By adding more check bits the code can be extended to test 16, 32, or more data bits. Hamming codes are fairly simple to implement. The code implementation uses XOR operations and lookup tables.

BCH codes have predetermined data lengths which they correct with a certain amount of check bits. A suitable code has to be chosen to cover the data which has to be checked. The (31,16) BCH code, for example, provides 3-bit correction for 16 bits of data with 15 check bits. This is more correction than the standard Hamming code can achieve, but the BCH code uses more check bits. These codes are implemented with a series of matrix calculations and are thus more complicated to compute.

## 5.3 Design of EDAC Circuit

This section covers the whole design of the EDAC for the ADSP-21061 processor. It starts with the Hamming code used for error checking and correction and then describes the host port, which was used to read and write to the processor's on-chip memory. The use of either a microprocessor or a FPGA to implement the EDAC code is discussed next, with different circuit configuration thereafter. The code used to implement the EDAC circuit and a simulation of the code is covered last.

### 5.3.1 Hamming Code

The Hamming code was chosen for its ease of implementation and adaptability. To calculate the Hamming code and the error syndrome, a series of XOR operations are used. Afterwards a lookup table is used to determine if there are any errors, and to correct

them. Both these functions are easy to implement on a microprocessor and a FPGA. The Hamming code is adaptable in the sense that the number of bits that are correctable can be adjusted by breaking the data into smaller pieces and generating a code for each. As an example, for a 32-bit block of data a single Hamming code can be generated so that one bit is correctable in the block, or it can be split into four 8-bit blocks where in each 8-bit block one bit is correctable.

Two cases of the Hamming code were considered for the EDAC circuit. One uses 5 check bits for 8-bit data and the other uses 6 check bits for 16-bit data. In both cases the code is able to detect two bit upsets and correct one bit upset, but not both simultaneously. More bits can be corrected per data block by reducing the length of the data block. For example 5 check bits can be used for 6 or 7 data bits. In our application each bit in memory has an even chance of being upset and in the LEO environment upsets happen once every few weeks according to calculations [15]. The 6-bit checksum Hamming code was chosen because it has sufficient EDAC capabilities for a LEO environment. The 32-bit data memory and 48-bit program memory is split into 16-bit blocks when the Hamming code is generated or checked.

The Hamming code used for this EDAC implementation is from the 74LS630 16-bit parallel EDAC circuit data sheet. To use the Hamming code for error detection or correction one first computes the Hamming check bits for the data. This is done by XOR' ing certain bits of the data together to form each check bit. Table 5.1 shows which data bits form each check bit. The check bits are stored together with the data.

**Table 5.1:** The table used to calculate the Hamming check bits from a 16-bit data word. The data bits marked with an "x" are XOR'ed together to form each check bit. [34]

| Check- | Bit number of data word | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| CB0 | x | x | | x | x | | | | x | x | x | | | x | | |
| CB1 | x | | x | x | | x | x | | x | | | x | | | x | |
| CB2 | | x | x | | x | x | | x | | x | | | x | | | x |
| CB3 | x | x | x | | | | x | x | | | x | x | x | | | |
| CB4 | | | | x | x | x | x | x | | | | | | x | x | x |
| CB5 | | | | | | | | | x | x | x | x | x | x | x | x |

To check the integrity of the data and check bits later, one calculates the Hamming check bits again from the data. The new check bits are then compared (XOR'ed) with the ones saved with the data. This produces the error syndrome, which is used to check for errors and correct them. A table is used to determine the condition of the data, which is either no errors, a specific data- or check-bit is inverted, or the data is uncorrectable. See Table 5.2 for the error syndrome lookup values. In the case of correctable errors the data bit which is in error is simply inverted to correct the data.

**Table 5.2:** The error syndrome lookup table used to determine the error type and location. [34]

| Inverted bit | Syndrome error code | | | | | |
|---|---|---|---|---|---|---|
| | CB0 | CB1 | CB2 | CB3 | CB4 | CB5 |
| No Error | 0 | 0 | 0 | 0 | 0 | 0 |
| DB0 | 1 | 1 | 0 | 1 | 0 | 0 |
| DB1 | 1 | 0 | 1 | 1 | 0 | 0 |
| DB2 | 0 | 1 | 1 | 1 | 0 | 0 |
| DB3 | 1 | 1 | 0 | 0 | 1 | 0 |
| DB4 | 1 | 0 | 1 | 0 | 1 | 0 |
| DB5 | 0 | 1 | 1 | 0 | 1 | 0 |
| DB6 | 0 | 1 | 0 | 1 | 1 | 0 |
| DB7 | 0 | 0 | 1 | 1 | 1 | 0 |
| DB8 | 1 | 1 | 0 | 0 | 0 | 1 |
| DB9 | 1 | 0 | 1 | 0 | 0 | 1 |
| DB10 | 1 | 0 | 0 | 1 | 0 | 1 |
| DB11 | 0 | 1 | 0 | 1 | 0 | 1 |
| DB12 | 0 | 0 | 1 | 1 | 0 | 1 |
| DB13 | 1 | 0 | 0 | 0 | 1 | 1 |
| DB14 | 0 | 1 | 0 | 0 | 1 | 1 |
| DB15 | 0 | 0 | 1 | 0 | 1 | 1 |
| CB0 | 1 | 0 | 0 | 0 | 0 | 0 |
| CB1 | 0 | 1 | 0 | 0 | 0 | 0 |
| CB2 | 0 | 0 | 1 | 0 | 0 | 0 |
| CB3 | 0 | 0 | 0 | 1 | 0 | 0 |
| CB4 | 0 | 0 | 0 | 0 | 1 | 0 |
| CB5 | 0 | 0 | 0 | 0 | 0 | 1 |
| Uncorrectable | All other combinations | | | | | |

### 5.3.2 ADSP-21061 Host Port

The ADSP-21061 host port is a special port which enables one to read from, and write to the internal SRAM of the processor. The high-speed operation of the processor is not influenced by host bus accesses because of FIFO buffers to the internal memory. Microprocessors can be interfaced to the host port with little additional hardware. One of the uses of the host port is for multi-processor configurations where the DSP processors have to access memory of other processors. This allows a processor to read from, and write to the memory of any of the processors on the bus.

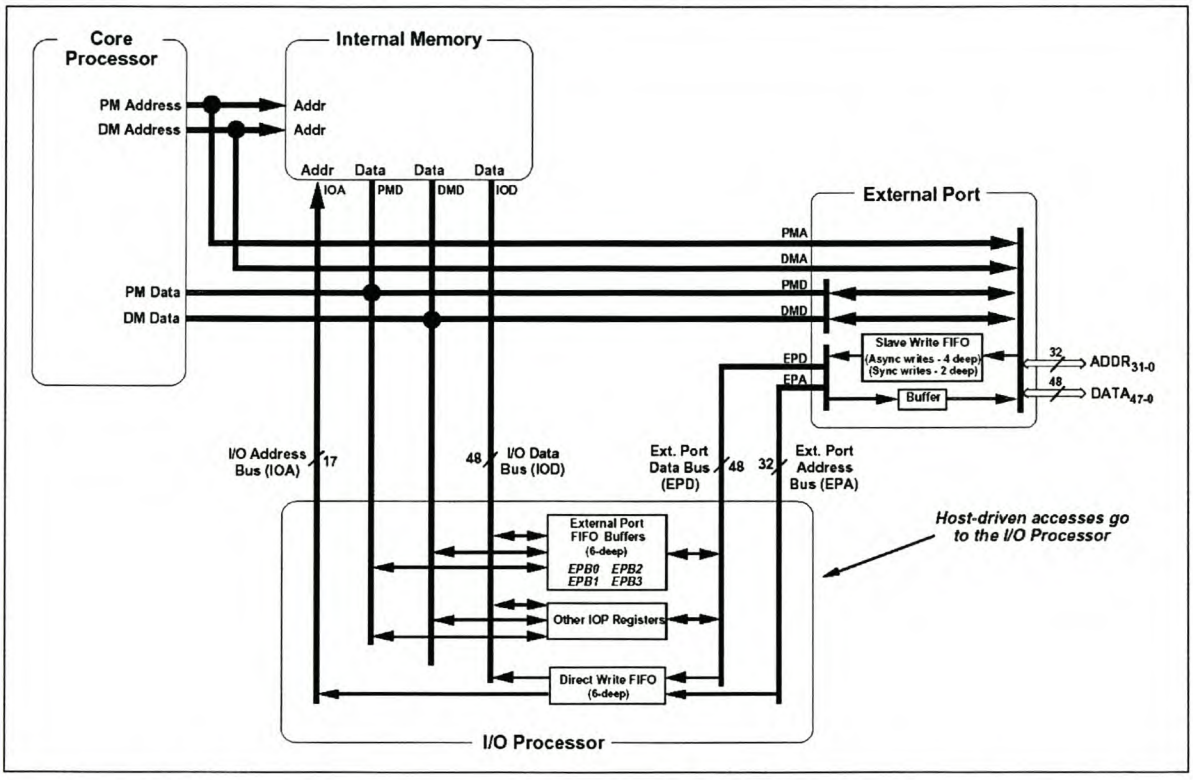



**Figure 5.1:** A diagram of the external port and host interface of the ADSP-21061 DSP processor. [1]

The host port is requested with the Host Bus Request (HBR) signal. The processor signals that the host bus is available by pulling the Host Bus Grant (HBG) signal low. This signals that the data and address bus, as well as the read (RD) and write (WR) pins are available to the external circuit. Interfacing to the internal memory is now done the same way as with general memory with these signals and buses.

The host bus can operate synchronously and asynchronously. Asynchronous interfacing is easier to use because the host does not have to run at the same high speed of the DSP processor. For asynchronous data transfers the host bus can be set up to be 16 or 32 bits wide. To read or write 32-bit data takes a single cycle with a 32-bit bus and two cycles with a 16-bit bus. Program memory, which is 48 bits wide, requires three read or write cycles for both 16- and 32-bit host buses.

### 5.3.3 Choice Between FPGA and Microprocessor

The host port interface of the ADSP-21061 makes it easy to interface a microprocessor or FPGA to the DSP processor. A microprocessor has the advantage that it is easy to program, and the hardware for interfacing to the ADSP-21061 host port is already built in. It is more difficult to use the FPGA because VHDL code has to be written which uses state machines to implement the host interface. The fact that data has to be written to, and read from the processor with the FPGA increases the state machine complexity. The generation and checking of the Hamming code is similar to implement with the VHDL and C programming languages.

The choice between the two was mainly determined by the speed at which they could calculate the Hamming codes and correct the data. A microprocessor has the limitation that it can only perform one operation at a time. Although the Hamming calculations mainly consist of simple XOR and shift operations on the processor, the number of them performed in series makes the microprocessor implementation slower. The procedure which calculates the check bits consisted of 130 lines of assembler code when compiled for the ADSP-2106x processor. The code was not optimised, so the number of operations could be reduced slightly. A FPGA can perform the XOR operations needed for the check bit generation in parallel, which allows them to be calculated in the shortest possible time. The calculation time then depends on the speed grade of the specific FPGA. In a test a FPGA took 21 ns to calculate the Hamming code and generate the syndrome. If optimised code took 100 operations to generate the same result on a processor, it would have to run at 4,7 GHz to achieve the same speed.

The FPGA was chosen in the end for its faster performance. The DSP processor runs at a high speed and one does not want to slow it down with a slow EDAC circuit.

### 5.3.4 EDAC Circuit Configurations

Various methods were considered for doing the internal memory checking of the DSP processor with the FPGA. The different implementations and their advantages and disadvantages are described below.

It is assumed that the PM will not change while the processor is running, so the Hamming codes for it have to be calculated just once. This allows the Hamming codes to be calculated beforehand, and uploaded together with the code. For space applications this is very handy when uploading new software to a satellite, because it ensures that the code is not corrupted in the process.

#### Checking with a FPGA Only

The first circuit considered was one where the FPGA uses the DSP host port to cycle through all the internal memory of the DSP. This configuration can be seen in Figure 5.2. It checks the Data Memory (DM) and Program Memory (PM) and corrects errors where possible. The problem with this method is that the DM changes constantly, so the processor has to indicate to the external circuit that specific memory has changed. The change indication can be done in the following ways:

1. If the Hamming codes are stored in DM locations, not all 32 bits are used. This allows the extra bits to be used to signal that the data for that Hamming code has changed. This means that each DM address requires another address to store the Hamming code and data change indication.

   To implement this method, the DSP writes 1's to the Hamming code location of the specific data word. This requires 2 extra lines of code on the DSP processor each time it writes new data to a memory location. The disadvantage of this method is that only 50% of the data memory is available for data storage, the rest is used for the Hamming codes and change bits.

2. Another solution is to store two Hamming codes in each DM location, and the change bits in separate memory. This saves more memory for general use. At least

two bits are necessary to indicate memory changes to ensure that upsets in these bits can be detected. This allows 16 change bits to be stored in one data word. The result is that for each 32-bit data word, half of a word (16 bits) will be used for the Hamming code and two bits for change indication.

To implement this method the DSP has to calculate the address of the change bits, as well as which bit positions to set in the word. This requires an overhead of 13 instructions on the DSP. The advantage is that 64% of the memory is available for data, 14% more than in the previous case. The amount of processor overhead is too much to make this solution practical however.

There are a few limitations and problems with this type of solution. One problem is that conflicts can occur when the DSP changes data while the FPGA is checking that data for errors or is already generating a new Hamming code for it. There is no simple method to solve this conflict. The method also cannot cover all the memory. The stack, for example, cannot be checked because it is changed by instructions that cannot be monitored. Due to these problems a better solution was sought, which is described below.

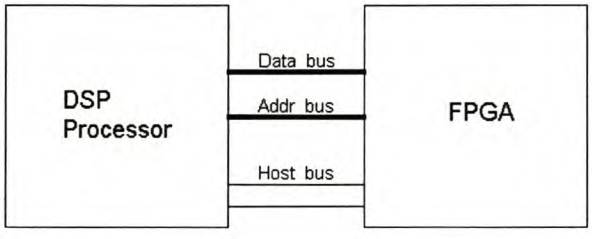



**Figure 5.2:** A diagram of the first EDAC circuit considered for the DSP processor

### Checking with a FPGA and an EDAC Circuit

The second method considered was to add external memory to the processor with a flow-through EDAC circuit in between. The external memory is defined as DM and a FPGA is added to check the internal memory, which is defined as PM. This setup can be seen in Figure 5.3.

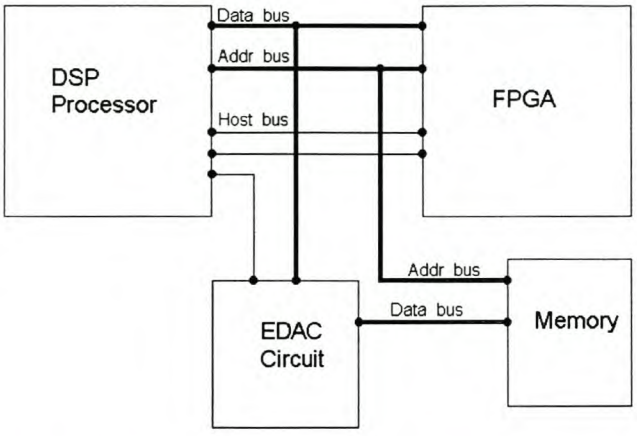



**Figure 5.3:** A diagram of the second EDAC configuration which uses a flow-through EDAC and FPGA

The check bits for two PM locations are stored in each PM word. A separate block of memory at the end of the DM is used to store all these words. This method allows 66% of the PM to be used for storing program code.

The host bus and external memory interface uses the same data and address bus, therefore only one of the two can be used at a time. This means the processor will have to wait when it wants to read or write external memory while the host bus is used. This setup ensures that no conflicts arise between the DSP processor writing new data and the EDAC circuit checking the same data. The program running on the processor is not affected by the external circuits. The speed of the DSP processor will depend on whether wait-states have to be added for the EDAC circuit and how long the FPGA uses the host bus.

### EDAC Circuit Implementation

The second method above was chosen as the best way to implement an EDAC circuit for the ADSP-21061 processor. It requires an extra component, but allows the processor to operate much faster and does not require any special code to be added to programs.

The operation of the EDAC circuit entails the continues checking of program memory. The Hamming code for 16 data bits was chosen, so the 48-bit PM is split up into three 16-bit blocks. The FPGA reads the 48 bits and then calculates and corrects the three

16-bit data blocks in parallel. This requires more logic in the FPGA, but is much faster. If any correctable upsets occurred in the memory read, the corrected values are written back to the DSP processor.

Registers were added to the FPGA to enable the DSP processor to set up the operation and read back the status of the EDAC circuit. The settings which can be configured are the start and end addresses of the block of memory which must be tested, the start address of the Hamming codes, and general settings. The status registers store the number of single-bit and double-bit upsets that have occurred. The registers are mapped to part of the external memory space, so the DSP processor simply has to read or write certain addresses.

A description of the VHDL code used for all these functions follows in the next section.

### 5.3.5 VHDL Code

The VHDL code can be split up into two parts. Synchronous processes were used to implement the state machine which controls the EDAC circuit operation. The Hamming calculations and memory interface signals were generated asynchronously. The asynchronous logic allows these signals to settle in the shortest time so that the maximum speed for the FPGA can be used. The synchronous and asynchronous code is discusses below. The full VHDL code listing is available in Appendix G.

#### Synchronous Code

The synchronous code mainly consists of a state machine which runs through a series of states to perform the EDAC functions. The main states with the states under them are:

- Wait_start
- Read_PM
  - Request_HB
  - Wait_HBG
  - Wait_REDY

  - Store_words
  - Release_HB
- Generate_ham
  - Calculate_ham
  - Check_data
  - Check_Hamming
- Write_PM
  - Request_HB
  - Write_words
  - Wait_HBG
  - Wait_REDY
  - Release_HB

The state machine starts at Wait_start and waits there until the EDAC circuit is enabled with the enable bit in the setup register. The device can be reset at any time with the reset pin. Once the EDAC circuit is enabled the state machine continues to the Read_PM state where it requests the host bus and reads 48 bits of memory. This is done with three consecutive 16-bit reads. The address it starts reading from is set up in the EDAC circuit registers and the type of memory (data or Hamming code) is determined by the Data_type variable. It always starts with data and thereafter reads the Hamming code. Once it has finished reading the host bus is released and the state machine moves on to Generate_ham.

In Generate_ham the state machine waits one clock cycle to make sure the error detection has finished. The Uncorrectable_error and Update_data signals are then checked to determine if the data is corrupt, and if not if it was corrected. If the data was corrupted the Double_errors_reg register is incremented and the main state returns to Wait_start. Otherwise if there were correctable upsets the state machine jumps to Write_PM to write the corrected data and returns. The Hamming code is checked next by testing the Update_Hamming signal. If it is set the state machine jumps to Write_PM to correct the Hamming code stored on the processor. The state machine returns to Wait_start hereafter.

The Write_PM state machine starts with requesting the host bus and then writes 48 bits PM to the processor with three write operations. The Data_type variable determines whether the data is written to the program– or Hamming code address. When it is finished it releases the host bus.

The Write_PM state machine uses the data type and address to determine if the data must be written to the Data_out or Hamming_out signals. The memory which stores Hamming codes contains two codes in each PM word. To determine which Hamming code is used for a specific PM location, the codes were ordered so that the lower codes are used for even addresses and the higher ones for uneven addresses. The VHDL code uses one address counter to read/write PM code and Hamming codes. This is done by using the setup registers to determine the number to add to the PM code address to get the Hamming code address. The lowest bit of the address then determines which of the two Hamming codes to use. The Read_PM state machine uses this same method. The address counter restarts from the PM_start_reg address when it reaches the PM_end_reg address.

### Asynchronous Code

The asynchronous part of the VHDL code consisted of signals which were not affected by the clock and ones that had to settle as fast as possible. The first code in the asynchronous part of the VHDL handles the interface signals with the host port. Certain values are assigned to these signals depending on whether the host bus is available and other criteria. Two address decoders are used to determine which of the internal registers of the EDAC circuit are addressed with the address port. This is combined with the interface logic to enable reading from and writing to them.

The biggest part of this section is used to calculate Hamming codes and syndromes, determine error signals, and correct data. These operations could be done synchronously as well, but would be slower because they would have to wait for the clock to start. The Hamming code calculations consist of a series of XOR operations as can be seen in the code listing in Appendix G. Three Hamming codes are calculated in parallel with the syndrome of each. The syndromes are used to generate signals which indicate the status of the data, namely if the data is correctable or not. If the data is correctable the corrected

data is calculated so it can be written back to the processor. For details on how the Hamming code works see Section 5.3.1.

### 5.3.6 Simulation of VHDL

The VHDL code was compiled and simulated using MAX+Plus II. A functional simulation was done to make sure the Hamming codes worked, all the registers functioned, and the addresses incremented correctly. From the simulations it could be seen that this worked well. The difficult part to simulate was the timings needed to interface to the host port. MAX+Plus II has a very simple simulator which is not suitable to use for large timing simulations. A better simulator would be needed to make sure the timings were correct. The simulations that were performed are in Appendix G.

# Chapter 6

# Conclusions

This chapter contains the conclusions reached after selecting DSP processors for future satellite use, performing SEU testing on one of the processors and designing an EDAC circuit for the internal memory of the processor. The chapter is split up into these three sections.

## 6.1 DSP Selection

A large number of processors were compared to find one suitable for use on a next satellite. The decision to use a floating-point processor reduced the possibilities to a few. The ADSP-2106x family from Analog Devices and TMS320C3x family from Texas Instruments were found to be the best processors to use currently for general satellite applications. From these families the ADSP-21061 and TMS320C31 processors were chosen because they were available on evaluation boards. The speed of DSP processors increases regularly and their power consumption fall. A new DSP processor selection list will have to be generated from time to time to compare the new processors.

General purpose processors are becoming faster and faster, and DSP instructions are being added to their instruction sets. This is making the gap between them and DSP processors smaller and smaller. In the future a DSP processor may be replaced with a general purpose one which has DSP instructions.

## 6.2 SEU Testing

The SEU tests were performed at two locations at the NAC and it was found that the proton therapy station was the better one of the two. The setup could be done much faster there because everything was in open air. The levels of proton energy and flux that were required for the tests could be measured there as well.

Similar methods and test programs to those used previously were implemented to do the different tests. These worked well and produced results. The low energy tests (45 MeV) for the cache and ALU programs did not deliver the predicted results, but higher cross-sections than expected. This was attributed to the processor not having time to recover from the 65 MeV tests done just before, or statistical variance in the measurements.

From the SEU test results it could be seen that the on-chip memory is much more sensitive to upsets than the rest of the processor. Predictions for the upset rate of the memory for a LEO environment will have to be calculated with programs to determine the sensitivity of the memory. If the internal memory of the processor is to be used, an EDAC circuit is needed to prevent errors due to upsets. The rest of the results compared well with previous tests and showed that the processor core will seldom experience upsets.

## 6.3 EDAC Implementation

Different configurations for implementing an EDAC circuit for the on-chip SRAM of the ADSP-21061 DSP processor were considered. The best solution was found to be one where external memory is added together with a flow-through EDAC circuit. A FPGA is used to check and correct the internal memory in this configuration. The external memory is used to store data while the internal is used for the code. This ensures that the Hamming codes for data that changed are generated and stored immediately. The operation of the FPGA was simulated and worked correctly. The circuit was not implemented, so the effect on the performance of the processor could not be determined.

# Bibliography

[1] ANALOG DEVICES. *ADSP-2106x SHARC User's Manual, Second Edition*, July 1996.

[2] ANALOG DEVICES. *ADSP-21061/ADSP-21061L Datasheet, Rev. A*, 1999.

[3] BERKELEY DESIGN TECHNOLOGY, I., "The BDTImark2000: A Measure of DSP Execution Speed." http://www.bdti.com/bdtimark/BDTImark2000.pdf. 2001.

[4] BERKELEY DESIGN TECHNOLOGY, I., "BDTImark2000 Scores." http://www.bdti.com/bdtimark/BDTImark2000graphic.pdf. June 2001.

[5] CARSON, M., DIZ, R., LONG, S., MACARTHUR, K., and TOTTY, K., "Radiation Hardening of Electronics." August 1997. http://dvorak.mse.vt.edu/faculty/hendricks/mse4206/projects97/group02/contents.htm.

[6] CRAIN, S., CRAIN, W., CRAWFORD, K., HANSEL, S., YU, P., and KOGA, R., "Single Event Effects Test Results for the 80C186 and 80C286 Microprocessors and the SMJ320C30 and SMJ320C40 Digital Signal Processors." *IEEE Transactions on Nuclear Science*, pp. 51–57.

[7] ELLIOTT, I., "Examples of VHDL Decriptions." 2000. http://www.ami.bolton.ac.uk/courseware/adveda/vhdl/vhdlexmp.html#Hamming Encoder.

[8] ENVIRONMENTS, E. S. and SECTION, E. A., "The Radiation Environment." http://www.estec.esa.nl/wmwww/WMA/rad_env.html. September 1998.

[9] ERICKSON, L., "Space Environment – Solar Radiation." September 2000. http://faculty.db.erau.edu/ericksol/courses/sp300/ch3/background_ch3.html.

[10] EYRE, J. and BIER, J., "DSP Processors Hit the Mainstream." *Computer*, August 1998, Vol. 31, No. 8, pp. 51–59. http://www.bdti.com/articles/final.pdf.

[11] GEORGE C. CLARKE, J. and CAIN, J. B., *Error-Correction Coding for Digital Communication*. New York: Plenum Press, 1981.

[12] GROBLER, H., "Aspects Affecting the Design of a Low Earth Orbit Satellite On-Board Computer." Master's thesis, University of Stellenbosch, December 2000.

[13] HAFFNER, J. W., *Radiation and Shielding in Space*, vol. 4 of *Nuclear Science and Technology*. Academic Press, 1967.

[14] HARBOE-SORENSEN, R., SERAN, H., P.ARMBRUSTER, and ADAMS, L., "The Single Event Upset Response of the Analog Devices, ADSP2100A, Digital Signal Processor." *IEEE Transactions on Nuclear Science*, 1992, pp. 441–445.

[15] HIEMSTRA, D. M. and BARIL, A., "Single Event Upset Characterization of the Pentium MMX and Pentium II Microprocessors using Proton Irradiation." *IEEE Transactions on Nuclear Science*, December 1999, Vol. 46, No. 6, pp. 1453–1460.

[16] HOLBERT, K., "Space Radiation Environmental Effects." http://www.eas.asu.edu/~holbert/eee460/spacerad.html. April 1997.

[17] "International Commission on Radiation Units and Measurements (ICRU) web page." http://www.icru.org. January 2002.

[18] JAMES, B. F., NORTON, O., and ALEXANDER, M. B., "The Natural Space Environment: Effects on Spacecraft." *NASA Scientific and Technical Information*, November 1994.

[19] JOHNSON, A., "Radiation Effects in Advanced Microelectronics Technologies." *IEEE Transactions on Nuclear Science*, 1998, Vol. 45, pp. 1339–1353.

[20] J.W. HOWARD, J. and HARDAGE, D., "Spacecraft Environments Interactions: Space Radiation and Its Effects on Electronic Systems." *NASA Scientific and Technical Information*, July 1999.

[21] KLEIN, J., "Source Code From Chapter 18 Of C Unleashed." http://home.att.net/~jackklein/C_Unleashed/hamming.c. 2000.

[22] KOGA, R., CRAWFORD, K., HANSEL, S., CRAIN, W., PENZIN, S., and MILLER, S., "The Risk of Utilizing SEE Sensitive COTS Digital Signal Processor (DSP) Devices in Space." *IEEE Transactions on Nuclear Science*, December 1996, Vol. 43, No. 6, pp. 2982–2989.

[23] LAPSLEY, P., BIER, J., SHOHAM, A., and LEE, E. A., *DSP Processor Fundamentals : architectures and features*. IEEE Press Series on Signal Processing. IEEE Press, 1997.

[24] LARIMER, J. and CHEN, D., "Fixed or floating? a pointed question in DSPs." *EDN Magazine*, August 1995.

[25] LAURIENTE, M. and VAMPOLA, A., "Spacecraft Anomalies due to Radiation Environment in Space." *NASDA/JAERI 2nd International Workshop on Radiation Effects of Semiconductor Devices for Space Applications*, March 1996.

[26] O'NEILL, P., BADHWAR, G., and CULPEPPER, W., "Risk Assessment for Heavy Ions of Parts Tested with Protons." *IEEE Transactions on Nuclear Science*, December 1997, Vol. 44, No. 6, pp. 2311–2314.

[27] PARKES, D. S., "DSP (Demanding Space-based Processing!): The Path Behind and the Road Ahead." *DSP '98, 6th International Workshop on Digital Signal Processing Techniques for Space Applications*, September 1998.

[28] PISACANE, V. L. and MOORE, R. C. (Eds), *Fundamentals of space systems*, Ch. Spacecraft reliability and quality assurance, pp. 690–717. The Johns Hopkins University : Applied Physics Laboratory series in science and engineering, New York, N.Y. : Oxford University Press, 1994.

[29] POIVEY, C., NOTEBAERT, O., GARNIER, P., CARRIERE, T., and BEAUCOUR, J., "Proton SEU Test of MC68020, MC68882, TMS320C25 on the ARIANE5 Launcher On Board Computer (OBC) and Inertial Reference System (SRI)." *IEEE Transactions on Nuclear Science*, June 1996, Vol. 43, No. 3, pp. 886–892.

[30] SHAEFFER, D., KIMBROUGH, J., WILLBURN, J., DENTON, S., KASCHMITTER, J., COLELLA, N., COAKLEY, P., and CASTENEDA, C.,

"Proton-Induced SEU, Dose Effects, and LEO Performance Predictions for R3000 Microprocessors." *IEEE Transactions on Nuclear Science*, December 1992, Vol. 39, No. 6, pp. 2309–2315.

[31] "Tutorial – Radiation Introduction." http://www.klabs.org/richcontent/Tutorial/Radiation_Introduction.htm. May 2001.

[32] SPRATT, J., PASSENHEIM, B., LEADON, R., CLARK, C. S., and STROBEL, D., "Effectiveness of IC Shielded Packages Against Space Radiation." *IEEE Transactions on Nuclear Science*, December 1997, Vol. 44, No. 6, pp. 2018–2025.

[33] SPRATT, J., PASSENHEIM, B., LEADON, R., CLARK, C. S., and STROBEL, D., "Effectiveness of IC Shielded Packages Against Space Radiation." *IEEE Transactions on Nuclear Science*, December 1997, Vol. 44, No. 6, pp. 2018–2025.

[34] TEXAS INSTRUMENTS. *Types SN54LS630, SN54LS631, SN74LS630, SN74LS631 16-bit parallel error detection and correction circuits*, March 1980.

[35] TEXAS INSTRUMENTS. *TMS320C31, TMS320LC31 Digital Signal Processors*, January 1999.

[36] VAN ROOYEN, D. M., "National Accelerator Centre." http://nac.ac.za. February 1999.

[37] WINOKUR, P., LUM, G., SHANEYFELT, M., SECTON, F., HASH, G., and SCOTT, L., "Use of COTS Microelectronics in Radiation Environments." *IEEE Transactions on Nuclear Science*, December 1999, Vol. 46, No. 6, pp. 1494–1503.

# Appendix A

# DSP Short List

This appendix lists the DSP processors that were considered for a future satellite. Each table covers a certain make of processor with horizontal lines distinguishing the families within that make. The processors are listed in the order Analog Devices, Motorola, Lucent, Texas Instruments and Temic.

**Table A.1:** The Analog Devices DSP processors short list.

| Processor | Supply Voltage (V) | Clock-Speed (MHz) | MFLOPS | MIPS | Fixed- / Floating-Point | Data Width (Bits) | Power Consumption | Ram (Program) | Rom | Used in space | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Analog Devices** | | | | | | | | | | | |
| ADSP-2101 | 5 V | | | 25 | Fixed | 16 | | | | | |
| ADSP-2103 | 3.3 V | | | 10.2 | Fixed | 16 | 65 mW max | | | | |
| ADSP-2104 | 3.3 V / 5 V | | | 13 / 20 | Fixed | 16 | | | | | |
| ADSP-2105 | 5 V | | | 20 | Fixed | 16 | | | | | |
| ADSP-2115 | 5 V | | | 25 | Fixed | 16 | | | | | |
| ADSP-2171 | 3.3 V / 5 V | | | 33 | Fixed | 16 | 300 mW typ. | | | AMSAT | |
| ADSP-2173 | 3.3 V | | | 20 | Fixed | 16 | 70 mW typ. | | | | |
| ADSP-2181 | 5 V | | | 40 | Fixed | 16 | | 80 kB | | | |
| ADSP-2183 | 3.3 V | | | 52 | Fixed | 16 | | 80 kB | | | |
| ADSP-2184 | 3.3 V / 5 V | 40 | | 40 | Fixed | 16 | 63 mA @ 33 MHz | 5 V | 20 kB | | |
| ADSP-2185 | 3.3 V / 5 V | | | 33 / 52 / 75 | Fixed | 16 | | 80 kB | | | |
| ADSP-2186 | 3.3 V / 5 V | | | 40 / 75 | Fixed | 16 | 0.4 mA / MIPS | 40 kB | | | |
| ADSP-2187L | 3.3 V | | | 52 | Fixed | 16 | | 160 kB | | | |
| ADSP-2188M | 2.5 V | | | 75 | Fixed | 16 | | 256 kB | | | |
| ADSP-2189M | 2.5 V | | | 75 | Fixed | 16 | 0.4 mA / MIPS | 192 kB | | | |
| AD14060 | 3.3 V / 5 V | | 480 | | | 32 | | | | | Quad DSP |
| AD14160 | 3.3 V / 5 V | | 480 | | | 32 | | | | | Quad DSP |
| ADSP-21020 | 5 V | 33,3 | | 33,3 | Floating | 32 | | 0 | 0 | ESA | Instr. Cache - 32 |
| ADSP-21060 | 3.3 V / 5 V | 40 | 120 | 40 | Floating | 32 | 670 mA high / 850 mA peak | 4 Mbit | | | Instr. Cache - 32 |
| ADSP-21061 | 3.3 V / 5 V | 50 | 150 | 50 | Floating | 32 | 670 mA high / 850 mA peak | 1 Mbit | | | Instr. Cache - 32 |
| ADSP-21062 | 3.3 V / 5 V | 40 | 120 | 40 | Floating | 32 | 670 mA high / 850 mA peak | 2 Mbit | | | Instr. Cache - 32 |
| ADSP-21065L | 3.3 V | 66 | 198 | 66 | Floating | 32 | 300 mA | | | | Instr. Cache - 32 |
| ADSP-21160M | 3.3 V | 100 | 600 | 100 | Floating | 32 | 940 mA @ 80 MHz | 2.5 V | | | |

**Table A.2:** The Motorola DSP processors short list.

| Processor | Supply Voltage (V) | Clock-Speed (MHz) | MFLOPS | MIPS | Fixed- / Floating-Point | Data Width (Bits) | Power Consumption | Ram (Program) | Rom | Used in space | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Motorola** | | | | | | | | | | | |
| DSP56002 | 5 V | 40 / 66 / 80 | | 20 / 33 / 40 | Fixed | 24 | | 256 w | 512 w | SUNSAT | |
| DSP56004 | 5 V | 50 / 66 / 81 | | 24 / 33 / 40.5 | Fixed | 24 | | 512 w | 512 w | | |
| DSP56007 | 5 V | 50 / 66 / 88 | | 25 / 33 / 44 | Fixed | 24 | 105 / 130 / 169 mA | 3200 w | 6400 w | | |
| DSP56009 | 5 V | 81 / 88 | | 40.5 / 44 | Fixed | 24 | | 512 w | 10 kw | | |
| DSP56301 | 3.3 V | 66 / 80 | | 66 / 80 | Fixed | 24 | 120 / 145 mA | 4 kw | - | | |
| DSP56303 | 3.3 V | 66 / 80 / 100 | | 66 / 80 / 100 | Fixed | 24 | 84 / 102 / 127 mA | 4 kw | - | | |
| DSP56307 | 2.5 V | 100 | | 100 | Fixed | 24 | 120 mA | 16 kw | - | | |
| DSP56309 | 3.3 V | 80 | | 80 | Fixed | 24 | | 20 kw | - | | |
| DSP56311 | 1.8 V | 150 | | 150 | Fixed | 24 | | 32 kw | - | | |
| DSP56362 | 3.3 V | 100 / 120 | | 100 / 120 | Fixed | 24 | 181 mA @ 100 MHz | 3 kw | 30 kw | | |
| DSP56364 | 3.3 V | 100 | | 100 | Fixed | 8 | | 512 w | 8 kw | | |
| DSP56366 | 3.3 V | 100 / 120 | | 100 / 120 | Fixed | 24 | | 3 kw | 40 kw | | |
| DSP56651 | 2.4 V | 58.8 | | 58.8 | Fixed | 16 | 35 mA typ. | 24 kw | 24 kw | | |
| DSP56652 | 1.8 V | 58.8 | | | Fixed | 16 | | 512 w | 48 kw | | |
| DSP56654 | 1.8 V | 58.8 | | | Fixed | 16 | | 40 kw | 2 kw | | |
| DSP56690 | 2.2 V | 104 | | | Fixed | 16 | | 3.5 kw | 84 kw | | |
| DSP56824 | 2.7 V | 70 | | 35 | Fixed | 16 | 30 mA max | 128 w | 32 kw | | |
| MSC8101 | 1.5 V | 300 | | 1200 | | 32 / 64 | | 256 kw | | | |

**Table A.3:** The Lucent DSP processors short list.

| Processor | Supply Voltage (V) | Clock-Speed (MHz) | MFLOPS | MIPS | Fixed- / Floating-Point | Data Width (Bits) | Power Consumption | Ram (Program) | Rom | Used in space | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Lucent** | | | | | | | | | | | |
| DSP1603F | | 33 | | 33 | Fixed | 16 | | x | x | | |
| DSP1604/06 | 3.3 V / 5 V | 33 | | 33 | Fixed | 16 | | x | x | | |
| DSP1607 | 3.3 V / 5 V | | | | Fixed | 16 | | x | x | | |
| DSP1608 | 5 V | | | 40 | Fixed | 16 | 9.5 mW / MIPS | | | | |
| DSP1609 | 3.3 V / 5 V | 70 | | 80 / 70 | Fixed | 16 | 6.5 mW / MIPS | x | x | | |
| DSP1615 | 3 V | | | 20 | Fixed | 16 | | 2 k | 24 k | | |
| DSP1616 | 2.7 V / 5 V | | | 26 / 50 | Fixed | 16 | | 2 k | 12 k | | |
| DSP1617 | 2.7 V / 5 V | | | 26 / 50 | Fixed | 16 | | 4 k | 24 k | | |
| DSP1618 | 2.7 V / 5 V | | | 26 / 50 | Fixed | 16 | | 4 k | 24 k | | |
| DSP1620 | 3 V / 5 V | | | 80 / 120 | Fixed | 16 | | 32 k | 4 k | | |
| DSP1625 | 2.7 V / 3 V | | | 80 / 100 | Fixed | 16 | | 8 k | 72 k | | |
| DSP1627 | 3 V / 5 V | | | 80 / 100 / 90 | Fixed | 16 | | 6 k | 36 k | | |
| DSP1628 | 3 V / 5 V | | | 80 | Fixed | 16 | | 16 k | 48 k | | |
| DSP1629 | 3 V / 5 V | | | 80 / 100 | Fixed | 16 | | 16 k | 48 k | | |
| DSP1650 | 3.3 V / 5 V | 40 | | 33 / 40 | Fixed | 16 | 3.3 / 9.5 mW / MIPS | x | x | | |
| DSP1653 | 5 V | 40 | | 40 | Fixed | 16 | | x | x | | |
| DSP1655 | 5 V | 40 | | 40 | Fixed | 16 | | x | x | | |
| DSP1656F | 3.3 V | | | 90 | Fixed | 16 | 6.5 mW / MIPS | | | | |
| DSP1660F | 3.3 V | | | 80 | Fixed | 16 | 4 mW / MIPS | | | | |
| DSP16210 | 2.5 V | 120 | | | Fixed | 16 | 1.2 mA / MIPS | | | | |
| DSP32C | | 12.5 / 16 / 20 | | | Floating | | | 1.5 / 2 k | - | | |

**Table A.4:** The Texas Instruments (Part 1) DSP processors short list.

| Processor | Supply Voltage (V) | Clock-Speed (MHz) | MFLOPS | MIPS | Fixed- / Floating-Point | Data Width (Bits) | Power Consumption | Ram (Program) | Rom | Used in space | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Texas Instruments** | | | | | | | | | | | |
| TMS320C25 | 5 V | 40 | | 10 | Fixed | 16 | | 544 w | 4 kw | KITSAT OSC-23 | |
| TMS320C30 | 5 V | 33 / 40 / 50 | 27 / 33.3 / 50 | 16.7 / 20 / 25 | Floating | 32 | 200 mA typ / 600 mA max | 2 kw | 4 kw | UOSAT, KITSAT | Cache - 64 |
| TMS320C31 | 3.3 V / 5 V | 50 / 60 / 80 | 50 / 60 / 80 | 25 / 30 / 40 | Floating | 32 | 275 mA typ / 550 mA max | 2 kw | - | UOSAT | Cache - 64 |
| TMS320C32 | 5 V | 40 / 50 / 60 | 40 / 50 / 60 | 20 / 25 / 30 | Floating | 32 | 225 mA typ / 475 mA max | 512 | - | | Cache - 64 |
| TMS320VC33 | 1.8 V / 3.3 V | 120 / 150 | | 60 / 75 | Floating | 32 | | 34 kw | - | | |
| TMS320C40 | 5 V | 50 / 60 | 50 / 60 | 25 / 30 | Floating | 32 | 850 / 950 mA | 2 kw | - | | Parallel |
| TMS320C44 | 5 V | 50 / 60 | 50 / 60 | 25 / 30 | Floating | 32 | 950 mA @ 60 MHz | 2 kw | - | | Processing |
| TMS320C50 | 5 V | 66 | | 33 | Fixed | 16 | | 9 kw | 2 kw | Yes | |
| TMS320LBC53 | | 40 | | 20 | Fixed | 16 | | 4 kw | 16 kw | | |
| TMS320C80 | 3.3 V | 50 | | 50 | | 32 | | 50 k | - | | 4× DSP + 1× RISC |
| TMS320C203 | | 40 / 57 / 80 | | 20 / 28.5 / 40 | Fixed | 16 | | 544 w | - | | |
| TMS320C206 | 3.3 V | 40 / 80 | | 20 / 40 | Fixed | 16 | 50 mA @ 40 MHz | 4.5 kw | 32 kw | | |
| TMS320C240 | 5 V | 20 | | 20 | Fixed | 16 | 80 mA typ. | 544 w | 16 kw | | |
| TMS320F241 | 5 V | 20 | | 20 | Fixed | 16 | 90 mA typ. | 544 w | - | | |
| TMS320C242 | 5 V | 20 | | 20 | Fixed | 16 | 100 mA typ. | | | | |
| TMS320F243 | 5 V | 20 | | 20 | Fixed | 16 | 120 mA typ. | 544 w | - | | |
| TMS320LC2402 | | 30 | | 30 | Fixed | 16 | | 544 w | 4 kw | | |

**Table A.5:** The Texas Instruments (Part 2) and Temic DSP processors short list.

| Processor | Supply Voltage (V) | Clock-Speed (MHz) | MFLOPS | MIPS | Fixed- / Floating-Point | Data Width (Bits) | Power Consumption | Ram (Program) | Rom | Used in space | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Texas Instruments** | | | | | | | | | | | |
| TMS320C541 | 3.3 V / 5 V | 40 / 66 | | 40 / 66 | Fixed | 16 | | 5 kw | 28 kw | | |
| TMS320C542 | 3.3 V / 5 V | 40 / 50 | | 40 / 50 | Fixed | 16 | | 10 kw | 2 kw | | |
| TMS320C543 | 3.3 V | 40 / 50 | | 40 / 50 | Fixed | 16 | 47 mA @ 40 MHz, 5 V | 10 kw | 2 kw | | |
| TMS320LC545 | 3.3 V | 40 / 50 / 66 | | 40 / 50 / 66 | Fixed | 16 | 30 mA @ 40 MHz | 6 kw | 48 kw | | |
| TMS320LC546 | 2.5 V / 3.3 V | 40 / 50 / 66 | | 40 / 50 / 66 | Fixed | 16 | | 6 kw | 48 kw | | |
| TMS320C549 | 2.5 V / 3.3 V | 80 / 100 / 120 | | 80 / 100 / 120 | Fixed | 16 | 20 mA @ 40 MHz | 32 kw | 16 kw | | |
| TMS320C55x | | | | 140–800 | Fixed | 16 | 0.05 mW / MIPS | | | | |
| TMS320C6205 | 1.5 V | 200 | | 200 | Fixed | 32 | | | | | |
| TMS320C6211 | 1.8 V | 150 | | 150 | Fixed | 32 | | | | | |
| TMS320C64x | | 600–1100 | | 4800–8800 | Fixed | 32 | | | | | |
| TMS320C6701 | 1.8 V | 150 / 167 | | 150 / 167 | Floating | 32 | 470 mA @ 150 MHz, 1,8 V | | | | |
| TMS320C6711 | 1.8 V | 100 / 150 | | 100 / 150 | Floating | 32 | | | | | |
| **Temic** | | | | | | | | | | | |
| TSC21020F | 5 V | 20 | 60 | 20 | Floating | 32 | 430 mA max | | | ESA | Radhard |

# Appendix B

# Radiation Hardware Schematics and Photos

This appendix contains the following:

- NAC layout diagram
- Schematic of all the hardware
- Photos of both control boards
- Photo of the brass block

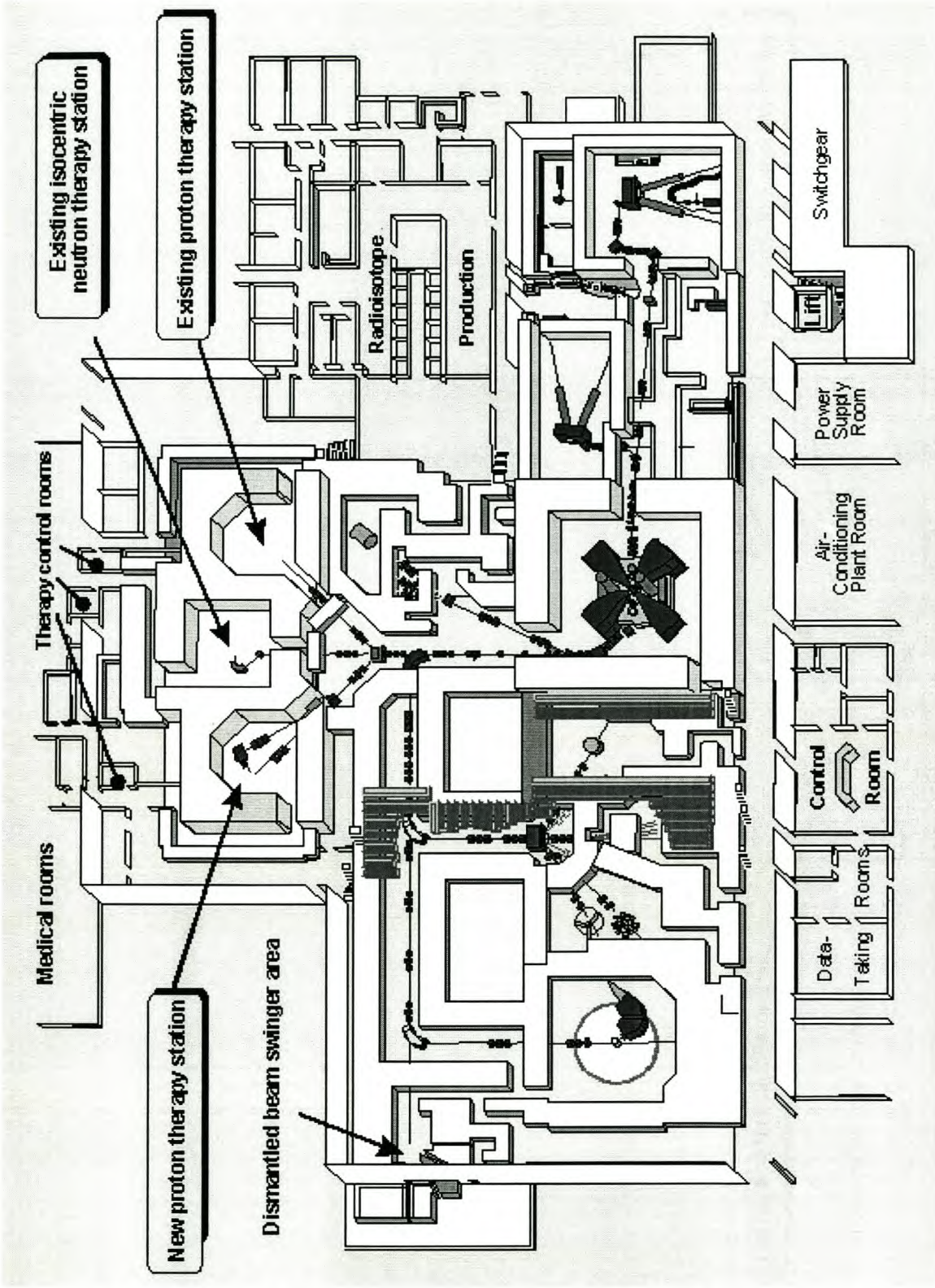



**Figure B.1:** A figure displaying the layout of the National Accelerator Centre (NAC). To the right SPC1 and SPC2 can be seen feeding the SSC. From there the radiation beam is steered to various rooms. (Courtesy of the NAC)

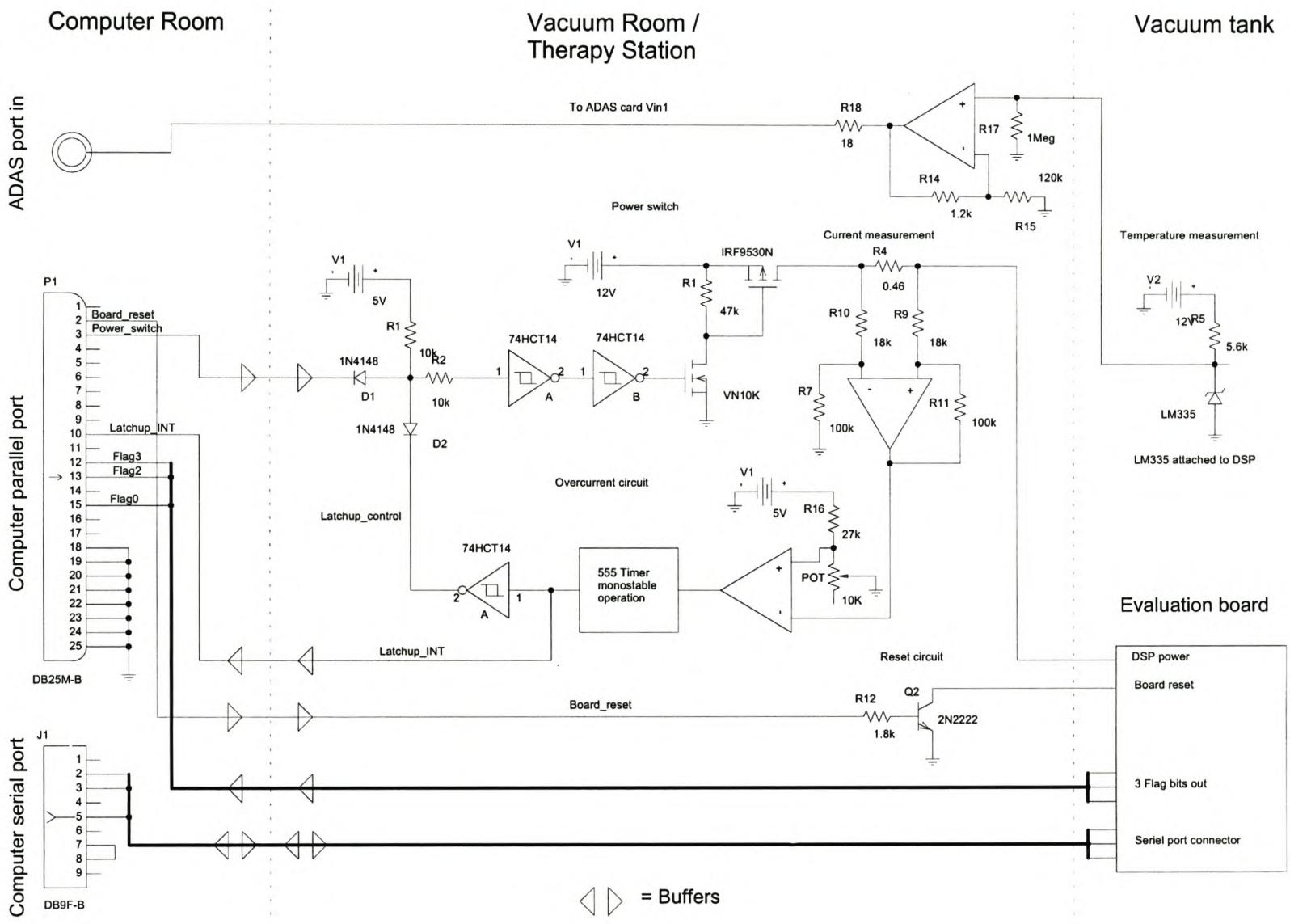



**Figure B.2:** A schematic of all the circuits used for the SEU testing setup.

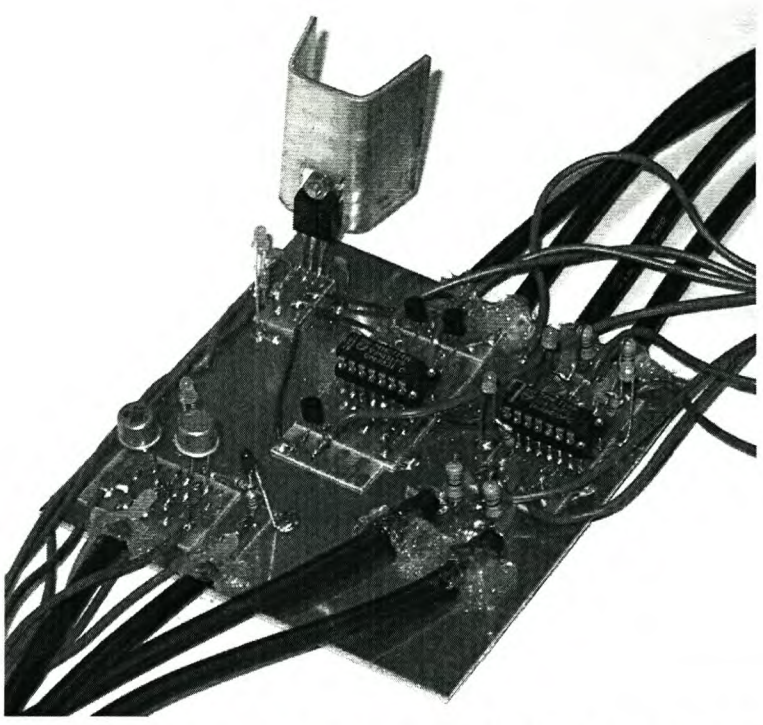


**Figure B.3:** A photo of the control board used for buffering signals to and from the computer.

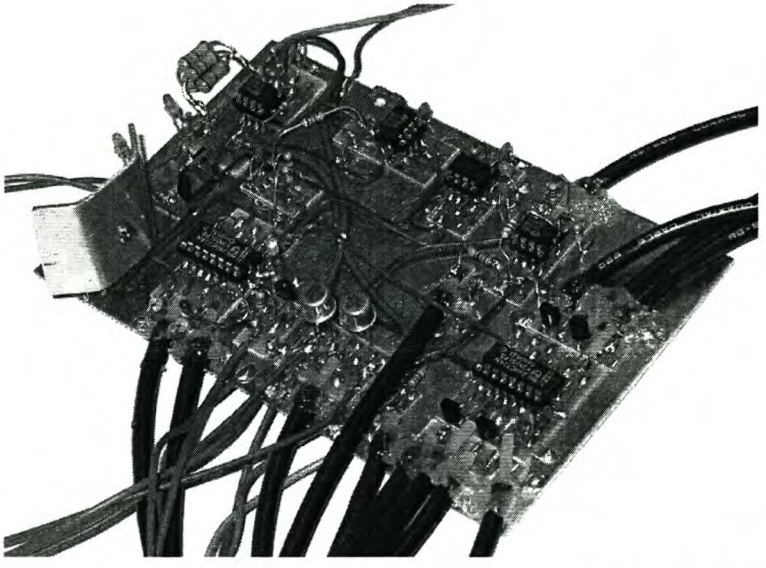


**Figure B.4:** A photo of the control board used to buffer signals to and from the computer room. A power switch and current limiting is implemented with the board as well.

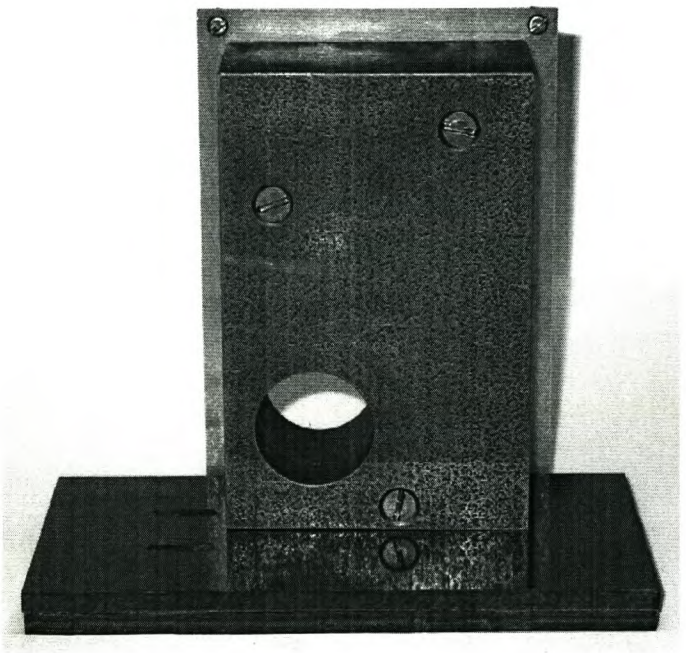


**Figure B.5:** A photo of the rear view of the brass block, with the hole through which the proton beam passes visible.

# Appendix C

# Proton Penetration Depth into Al and Brass

This appendix contains a graph of proton energy versus the penetration depth of protons into Aluminium (Al) and brass. These numbers were calculated by a computer program at the NAC and are available in table format in ICRU Report 49, "Stopping Power and Ranges for Protons and Alpha Particles" [17] as well.

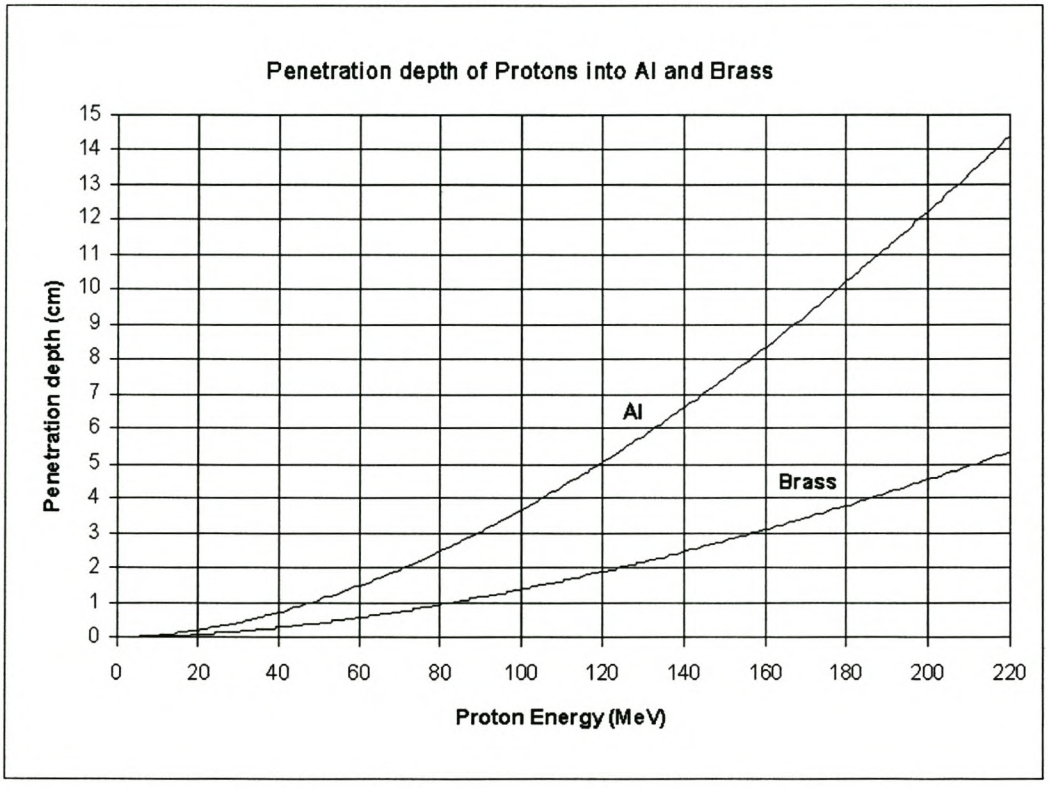



**Figure C.1:** A graph of proton energy versus the penetration depth of protons into Aluminium (Al) and Brass.

# Appendix D

# NAC Specifications

Detail specifications and information on the National Accelerator Centre (NAC) is tabled in this appendix.

**Table D.1:** Details of the NAC. [36]

| **History** | | SPC 1 | SPC 2 | SSC |
|---|---|---|---|---|
| Milestone dates | Design | 1978 | 1986 – 1992 | 1977 |
| | Model tests | 1979 – 1980 | - | - |
| | Construction | 1978 – 1983 | 1989 – 1993 | 1979 |
| | First beam | December 1983 | December 1993 | October 1985 |
| Design / Construction by | | in house with various engineering contractors | in house; various contracts | in house; various contracts |
| **Magnet** | | SPC 1 | SPC 2 | SSC |
| Pole parameters | diameter | 116 cm | 116 cm | - |
| | $R_{extract}$ | 47.6 cm | 47.6 cm | 443 cm |
| | $R_{inject}$ | - | - | 101 cm |
| Hill parameters | gap (max) | 15.6 cm | 15.6 cm | 6.6 cm |
| | $B_{min}$ | 1.23 T | 1.24 T | 1.27 T |
| Valley parameters | gap (max) | 25.0 cm | 25.0 cm | |
| | $B_{min}$ | 0.75 T | 0.75 T | |
| Average field | $< B >_{min}$ | 0.3 T | 0.3 T | - |
| | $< B >_{max}$ | 0.98 T | 1.0 T | 0.24 T |
| Number of sectors | compact / separated | 4 / - | 4 / - | - / 4 |
| | sector angle | 45° | 45° | 34° |
| | spiral (max) | - | - | 0° |
| Field trimming | Trim coils | 5 | 6 | 29 |
| | Harmonic coils | 2 | 2 | 0 |
| | Other | 2 Cone coils | 2 Cone coils | 2 yoke coils on each |
| | | | | 4 sectors |
| Current | Main coils | 800 A | 690 A | 1600 A |
| | Stability | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ |
| | Trim coils | 180 A | 200 A | 500 A |
| | Stability | $10^{-4}$ | $10^{-3}$ | $10^{-4}$ |
| Weight | Iron | 54.5 tons | 54.5 tons | 1400 tons |
| | Conductor | 1.85 tons copper | 1.85 tons copper | 5.8 tons |
| Ion energy | Bending limit, E/A | 11 $q^2/A^2$ $MeV/u$ | 11 $q^2/A^2$ $MeV/u$ | 220 $q^2/A^2$ $MeV/u$ |
| | Focusing limit, E/A | - | - | 220 q/A MeV/u |
| **Acceleration system** | | SPC 1 | SPC 2 | SSC |
| Fundamental acceleration | Description | Two $\lambda/4$ resonators with 90° dees | Two $\lambda/4$ resonators with 90° dees | Two $\lambda/2$ resonators, push-push mode |
| | No. of gaps/turn | 4 | 4 | 4 |
| | dE/dn (max) | 0.24 MeV/q | 0.24 MeV/q | 1 MeV/q |
| | Voltage (max) | 0.060 MV | 0.060 MV | 0.25 MV |
| | Harmonic $f_{rf}$ / $f_{ion}$ | 2, 6 | 2, 6 | 4, 12 |
| | Frequency | 8.6 – 26 MHz | 8.6 – 26 MHz | 6 – 26 MHz |
| | Power in (max) | 2 × 0.025 MW | 2 × 0.025 MW | 2 × 0.15 MW |
| | Stability, phase | 0.1° | 0.1° | 0.1° |
| | Voltage | $10^{-3}$ | $10^{-3}$ | $10^{-3}$ |
| Other Cavities | Flattopping | None | None | None |

**Table D.1:** Continued...

| **Vacuum system** | | SPC 1 | SPC 2 | SSC |
|---|---|---|---|---|
| | Operating pressure | $1.5 \times 10^{-5}$ mbar | $2 \times 10^{-7}$ mbar | $7 \times 10^{-7}$ mbar |
| | Pumps | turbo 4.58 $m^3/s$<br>roots 350 $m^3/h$<br>rotary vane 60 $m^3/h$ | turbo 2.2 $m^3/s$<br>cryo 10 $m^3/h$<br>2 LN2 cryo 10 $m^3/h$<br>rotary vane 60 $m^3/h$ | 6 turbo 2 $m^3/s$<br>2 cryo 5 $m^3/h$<br>2 LN2 cryo 10 $m^3/h$<br>4 rotary vane 120 $m^3/h$<br>4 roots 350 $m^3/h$ |
| **Ion source(s)** | | SPC 1 | SPC 2 | SSC |
| | Type | PIG | ECR | NA |
| | Intensity ($\mu A$) | $4 \times 10^3$ | differs widely | |
| | Emittance | - | - | |
| | Ions species | p | p to Xe | |
| | Type | | Atomic beam polarised ion source | |
| | Intensity ($\mu A$) | | 28 | |
| | Emittance, $\pi$ mm mrad | | 0.9 | |
| | Ion species | | p, d | |
| **Extraction and injection system** | | SPC 1 | SPC 2 | SSC |
| Injection system | Type | NA | 3 interchangeable spiral inflectors | 2 dipoles and a magnetic channel |
| | Efficiency | NA | A: 40%, B: 50%, C: 30% | 100% |
| Extraction system | Type | Electrostatic and 2 magnetic channels | Electrostatic and 2 magnetic channels | 1 electrostatic channel and 2 septum magnets |
| | Efficiency | 96% | A: 75%, B: 80%, C: 50% | 100% |
| **Beam information** | | SPC 1 | SPC 2 | SSC |
| Extracted beam properties | For beam current ($\mu$A) | 230 | | 35 |
| | MeV / u | 3.14 | | 66 |
| | Particle | p | | p |
| | $\Delta$ E / E () | 0.85 | | 0.4 |
| | $\Delta\phi$ (°rf) | 15 | | 10 |
| | $\varepsilon_n = \beta\gamma\varepsilon$ ($\pi$ mm mrad) | 0.98 | | 2.7 |
| | z ($\pi$ mm mrad) | 0.49 | | 0.8 |

# Appendix E

# DSP Processor Test Software

The flow diagrams and code listings of the programs used on the DSP processor for radiation testing are listed in this appendix. These include the NOP, cache and ALU tests.

## E.1 Flow Diagrams of DSP Test Programs

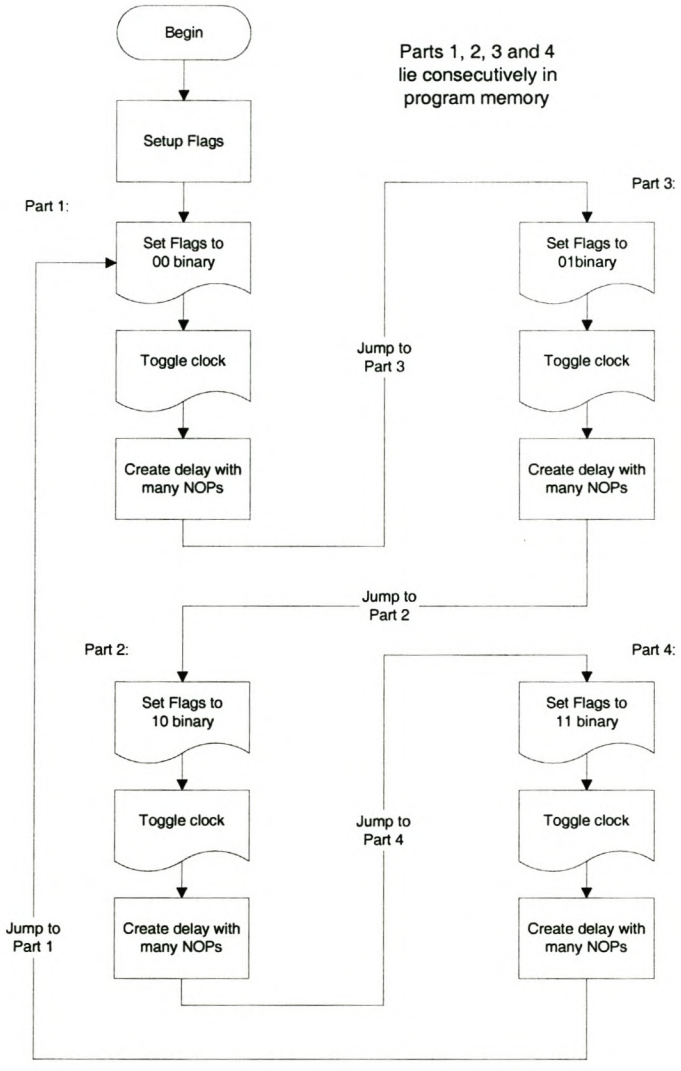



**Figure E.1:** A flow diagram of the NOP test program which tests the susceptibility of the instruction pointer to upsets.

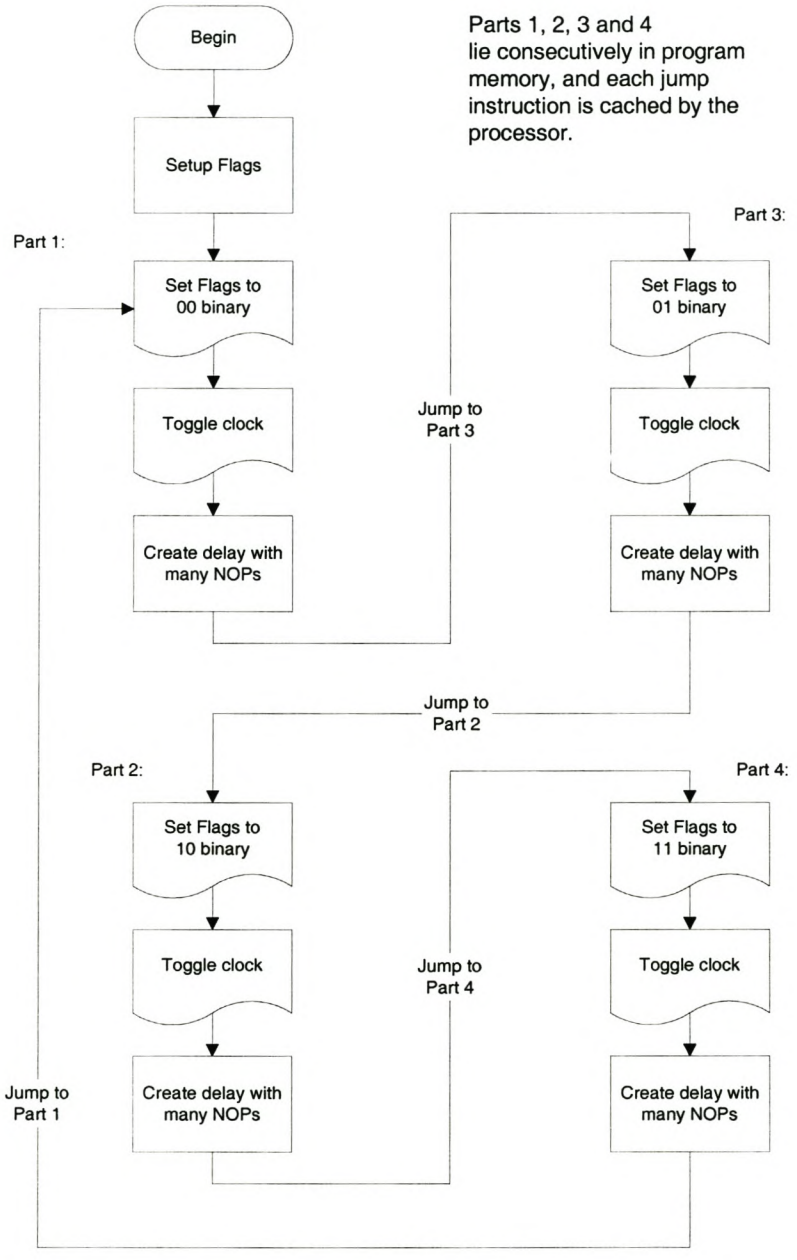



**Figure E.2:** A flow diagram of the Cache test program which tests the susceptibility of the cache to upsets.

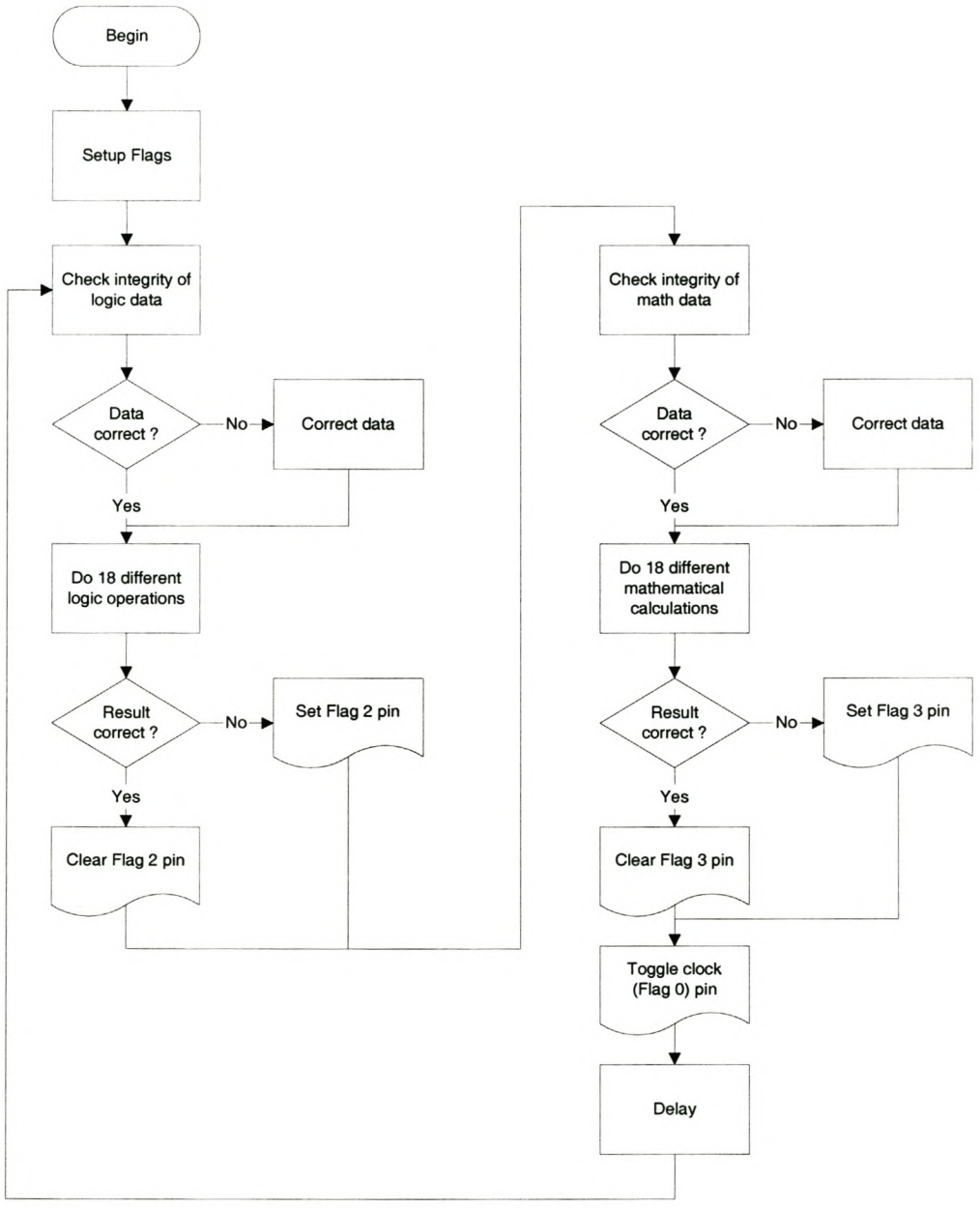



**Figure E.3:** A flow diagram of the ALU test program which tests the susceptibility of the mathematical hardware to upsets.

## E.2 Shortened Code for NOP Test

```
/*********************************************************/
/*                                                       */
/*   NOP.asm                                             */
/*                                                       */
/*   Program to test if the program counter increments   */
/*   correctly during radiation                          */
/*                                                       */
/*********************************************************/

   #define FLAGBIT0 0x00080000
   #define FLAGBIT2 0x00200000
   #define FLAGBIT3 0x00400000


.SEGMENT/PM seg_pmco;


   BIT CLR MODE1 0x00000100;          /* Setup mode2 register so flags are outputs */
   BIT CLR MODE2 0x00010020;
   BIT SET MODE2 0x00068010;

   BIT CLR ASTAT FLAGBIT3;            /* Set the flags for startup */
   BIT CLR ASTAT FLAGBIT2;
   BIT CLR ASTAT FLAGBIT0;

/*********************************************************/
/*                                                       */
/*   The first block of the loop                          */
/*                                                       */
/*********************************************************/

   block1 :

      BIT CLR ASTAT FLAGBIT2;         /* Set the flags ( 0 ) */
      BIT CLR ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;         /* Toggle clock to signal new data */

      nop;
      nop;
      nop;

      /* total of 760 NOP's */

      nop;
      nop;
      nop;

      JUMP block3;                    /* Jump to block 3 */

/*********************************************************/
/*                                                       */
/*   The second block of the loop                         */
/*                                                       */
/*********************************************************/

   block2 :

      BIT CLR ASTAT FLAGBIT2;         /* Set the flags ( 2 ) */
      BIT SET ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;         /* Toggle clock to signal new data */

      nop;
      nop;
      nop;

      /* total of 760 NOP's */

      nop;
      nop;
```

```
      nop;

      JUMP block4;                    /* Jump to block 4 */

/*********************************************************/
/*                                                       */
/*   The third block of the loop                          */
/*                                                       */
/*********************************************************/

   block3 :

      BIT SET ASTAT FLAGBIT2;         /* Set the flags ( 1 ) */
      BIT CLR ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;         /* Toggle clock to signal new data */

      nop;
      nop;
      nop;

      /* total of 760 NOP's */

      nop;
      nop;
      nop;

      JUMP block2;                    /* Jump to block 2 */

/*********************************************************/
/*                                                       */
/*   The fourth block of the loop                         */
/*                                                       */
/*********************************************************/

   block4 :

      BIT SET ASTAT FLAGBIT2;         /* Set the flags ( 3 ) */
      BIT SET ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;         /* Toggle clock to signal new data */

      nop;
      nop;
      nop;

      /* Total of 760 NOP's */

      nop;
      nop;
      nop;

      JUMP block1;                    /* Jump to start */

      nop;
      nop;
      nop;
      nop;
      nop;

.ENDSEG;       // End of code segment
```

## E.3 Shortened Code for Cache Test

```
/*********************************************************/
/*                                                       */
/*   Cache.asm                                           */
/*                                                       */
/*   Program to test if the program counter increments   */
/*   correctly during radiation, while code is in cache  */
/*                                                       */
/*********************************************************/

   #define FLAGBIT0 0x00080000
   #define FLAGBIT2 0x00200000
   #define FLAGBIT3 0x00400000


.SEGMENT/PM seg_pmco;

   /* Setup mode2 register so flags are outputs */

   BIT CLR MODE1 0x00000100;
   BIT CLR MODE2 0x00010020;
   BIT SET MODE2 0x00068010;

   BIT CLR ASTAT FLAGBIT3;              /* Set the flags for startup */
   BIT CLR ASTAT FLAGBIT2;
   BIT CLR ASTAT FLAGBIT0;

/*********************************************************/
/*                                                       */
/*   The first part of the loop                          */
/*                                                       */
/*********************************************************/

   part1 :

      BIT CLR ASTAT FLAGBIT2;           /* Set the flags ( 0 ) */
      BIT CLR ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;           /* Toggle clock to signal new data */

      nop;
      nop;
      nop;

      /* total of 760 NOP's */

      nop;
      nop;
      nop;

      r0 = pm( 0x00020400 );            /* Dummy read to cache the Jump */
      nop;

      JUMP part3;                       /* Jump to part 3 */

/*********************************************************/
/*                                                       */
/*   The second part of the loop                         */
/*                                                       */
/*********************************************************/

   part2 :

      BIT CLR ASTAT FLAGBIT2;           /* Set the flags ( 2 ) */
      BIT SET ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;           /* Toggle clock to signal new data */

      nop;
      nop;
      nop;
```

```
      /* total of 760 NOP's */

      nop;
      nop;
      nop;

      r0 = pm( 0x00020400 );          /* Dummy read to cache the Jump */
      nop;

      JUMP part4;                     /* Jump to part 4 */

/*********************************************************/
/*                                                       */
/*   The third part of the loop                          */
/*                                                       */
/*********************************************************/

   part3 :

      BIT SET ASTAT FLAGBIT2;         /* Set the flags ( 1 ) */
      BIT CLR ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;         /* Toggle clock to signal new data */

      nop;
      nop;
      nop;

      /* total of 760 NOP's */

      nop;
      nop;
      nop;

      r0 = pm( 0x00020400 );          /* Dummy read to cache the Jump */
      nop;

      JUMP part2;                     /* Jump to part 2 */

/*********************************************************/
/*                                                       */
/*   The fourth part of the loop                         */
/*                                                       */
/*********************************************************/

   part4 :

      BIT SET ASTAT FLAGBIT2;        /* Set the flags ( 3 ) */
      BIT SET ASTAT FLAGBIT3;
      BIT TGL ASTAT FLAGBIT0;        /* Toggle clock to signal new data */

      nop;
      nop;
      nop;

      /* Total of 760 NOP's */

      nop;
      nop;
      nop;

      r0 = pm( 0x00020400 );          /* Dummy read to cache the Jump */
      nop;

      JUMP part1;                     /* Jump to start */

      nop;
      nop;
      nop;
      nop;
      nop;

.ENDSEG;       // End of code segment
```

## E.4 Code for ALU Test

```
/*************************************************************/
/*                                                           */
/*  ALU3.c          16-03-2001                               */
/*                                                           */
/*  Program that does Logic and Math calculations, and       */
/*  then checks if the results are correct. The Flags are    */
/*  cleared for a right result, and set for a wrong one.     */
/*                                                           */
/*************************************************************/

asm ("
#define FLAGBIT0 0x00080000             /* define flag bits */
#define FLAGBIT2 0x00200000
#define FLAGBIT3 0x00400000 ");

/*************************************************************/

                                          /* start values with extras */
   unsigned long Logic_start_value0 = 0x2B3AB2F2,
                 Logic_start_value1 = 0x2B3AB2F2,
                 Logic_start_value2 = 0x2B3AB2F2,
                                            /* end values with extras */
                 Logic_end_value0 = 0xF58EF602,
                 Logic_end_value1 = 0xF58EF602,
                 Logic_end_value2 = 0xF58EF602,

                 Logic_test,                  /* variable for calculations */

                 I;                                  /* general counter */

                                          /* start values with extras */
   float         Math_start_value0 = 12.45,
                 Math_start_value1 = 12.45,
                 Math_start_value2 = 12.45,
                                            /* end values with extras */
                 Math_end_value0 = 1.33991516113281e2,
                 Math_end_value1 = 1.33991516113281e2,
                 Math_end_value2 = 1.33991516113281e2,

                 Math_test;                   /* variable for calculations */

void main ( void )
{
   asm ("
   BIT CLR ASTAT FLAGBIT0;                 /* Clear the flags for startup */
   BIT CLR ASTAT FLAGBIT2;
   BIT CLR ASTAT FLAGBIT3; ");

   while( 1 )                                          /* endless loop */
   {

   /* Start Logic test */

                                  /* check integrity of Logic start data */
      if ( Logic_start_value0 != Logic_start_value1 )
      {
         if ( Logic_start_value0 != Logic_start_value2 )
         {
            if ( Logic_start_value1 != Logic_start_value2)
               while( 1 );                /* Data corrupt, create timeout */
            else Logic_start_value0 = Logic_start_value1;
         }
         else Logic_start_value1 = Logic_start_value0;
      }
      else if ( Logic_start_value2 != Logic_start_value0)
         Logic_start_value2 = Logic_start_value0;

                                    /* check integrity of Logic end data */
      if ( Logic_end_value0 != Logic_end_value1 )
```

```
{
   if ( Logic_end_value0 != Logic_end_value2 )
   {
      if ( Logic_end_value1 != Logic_end_value2)
         while( 1 );                  /* Data corrupt, create timeout */
      else Logic_end_value0 = Logic_end_value1;
   }
   else Logic_end_value1 = Logic_end_value0;
}
else if ( Logic_end_value2 != Logic_end_value0)
   Logic_end_value2 = Logic_end_value0;

Logic_test = Logic_start_value0;              /* assign start values */

                                             /* do logical operations */
asm("
   r2=dm(_Logic_test);

   r4=-841690679;
   r2=r2 and r4;

   r2=lshift r2 by -2;

   r4=1453303568;
   r2=r2 or r4;

   r2=not r2;

   r2=lshift r2 by 3;

   r4=1781371845;
   r2=r2 xor r4;

   r4=1654630715;
   r2=r2 and r4;

   r2=lshift r2 by -1;

   r4=-979417168;
   r2=r2 or r4;

   r2=not r2;

   r2=lshift r2 by 1;

   r4=393066949;
   r2=r2 xor r4;

   r4=1962359221;
   r2=r2 and r4;

   r2=lshift r2 by -1;

   r4=1737116963;
   r2=r2 or r4;

   r2=not r2;

   r2=lshift r2 by 2;

   r4=-1793441934;
   r2=r2 xor r4;

   dm(_Logic_test)=r2;  ");

if ( Logic_test != Logic_end_value0 )
   asm (" BIT SET ASTAT FLAGBIT2; ");          /* set flag for error */
else
   asm (" BIT CLR ASTAT FLAGBIT2; ");           /* clear flag for OK */


/* Start Math test */

                              /* check integrity of Math start data */
```

```
if ( Math_start_value0 != Math_start_value1 )
{
   if ( Math_start_value0 != Math_start_value2 )
   {
      if ( Math_start_value1 != Math_start_value2)
         while( 1 );                  /* Data corrupt, wait for reset */
      else Math_start_value0 = Math_start_value1;
   }
   else Math_start_value1 = Math_start_value0;
}
else if ( Math_start_value2 != Math_start_value0)
   Math_start_value2 = Math_start_value0;

                                 /* check integrity of Math end data */
if ( Math_end_value0 != Math_end_value1 )
{
   if ( Math_end_value0 != Math_end_value2 )
   {
      if ( Math_end_value1 != Math_end_value2)
         while( 1 );                  /* Data corrupt, wait for reset */
      else Math_end_value0 = Math_end_value1;
   }
   else Math_end_value1 = Math_end_value0;
}
else if ( Math_end_value2 != Math_end_value0)
   Math_end_value2 = Math_end_value0;


Math_test = Math_start_value0;                  /* assign start value */

                                                /* do math operations */
asm("
   f8=dm( _Math_test );

   f12= 0x423570a4;
   f2=f8+f12;

   f4= 0x404d70a4;
   f2=f2*f4;

   f12= 0x416dcac1;
   f2=f2-f12;

   f4=f2;
   f8= 0x40000000;
   f5= 0x41877ae1;
   f2=recips f5;
   f12=f2*f5;
   f4=f2*f4, f2=f8-f12;
   f12=f2*f12;
   f4=f2*f4, f2=f8-f12;
   f12=f2*f12;
   f4=f2*f4, f2=f8-f12;
   f4=f2*f4;
   f2=f2-f12;
   f2=f2*f4;
   f2=f2+f4;

   f12= 0x3e969446;
   f2=f2+f12;

   f4= 0x40e9fbe7;
   f2=f2*f4;

   f12= 0x41c7a5e3;
   f2=f2-f12;

   f4=f2;
   f8= 0x40000000;
   f5= 0x40925604;
   f2=recips f5;
   f12=f2*f5;
   f4=f2*f4, f2=f8-f12;
   f12=f2*f12;
```

```
    f4=f2*f4, f2=f8-f12;
    f12=f2*f12;
    f4=f2*f4, f2=f8-f12;
    f4=f2*f4;
    f2=f2-f12;
    f2=f2*f4;
    f2=f2+f4;

    f12= 0x458cb666;
    f2=f2+f12;

    f4= 0x3eb6c8b4;
    f2=f2*f4;

    f12= 0x42cee666;
    f2=f2-f12;

    f4=f2;
    f8= 0x40000000;
    f5= 0x41877ae1;
    f2=recips f5;
    f12=f2*f5;
    f4=f2*f4, f2=f8-f12;
    f12=f2*f12;
    f4=f2*f4, f2=f8-f12;
    f12=f2*f12;
    f4=f2*f4, f2=f8-f12;
    f4=f2*f4;
    f2=f2-f12;
    f2=f2*f4;
    f2=f2+f4;

    f12= 0x43c3b99a;
    f2=f2+f12;

    f4= 0x3faf9db2;
    f2=f2*f4;

    f12= 0x431c06a8;
    f2=f2-f12;

    f4=f2;
    f8= 0x40000000;
    f5= 0x4021374c;
    f2=recips f5;
    f12=f2*f5;
    f4=f2*f4, f2=f8-f12;
    f12=f2*f12;
    f4=f2*f4, f2=f8-f12;
    f12=f2*f12;
    f4=f2*f4, f2=f8-f12;
    f4=f2*f4;
    f2=f2-f12;
    f2=f2*f4;
    f2=f2+f4;

    f12= 0x416efdf4;
    f2=f2+f12;

    f12= 0x42a16d91;
    f2=f2-f12;

    dm( _Math_test )=f2;   ");

if ( Math_test != Math_end_value0 )
    asm (" BIT SET ASTAT FLAGBIT3; ");          /* set flag for error */
else
    asm (" BIT CLR ASTAT FLAGBIT3; ");           /* clear flag for OK */

asm (" BIT TGL ASTAT FLAGBIT0; ");      /* Flag0 shows new data ready */

for ( I = 0; I != 28; I++ )                          /* create delay */
    asm (" nop;
           nop;
```

```
                nop;
                nop;
                nop; ");

    }   /* while */

  return;

}   /* main */

/*********************************************************/
```

# Appendix F

# Computer Test Software

This appendix contains the flow diagrams and code of the test procedures that were run on the computer. The Pascal code for the NOP, cache, ALU and memory procedures are listed. A flow diagram for the memory test is included as well.

# F.1 Flow Diagram of Memory Test Program

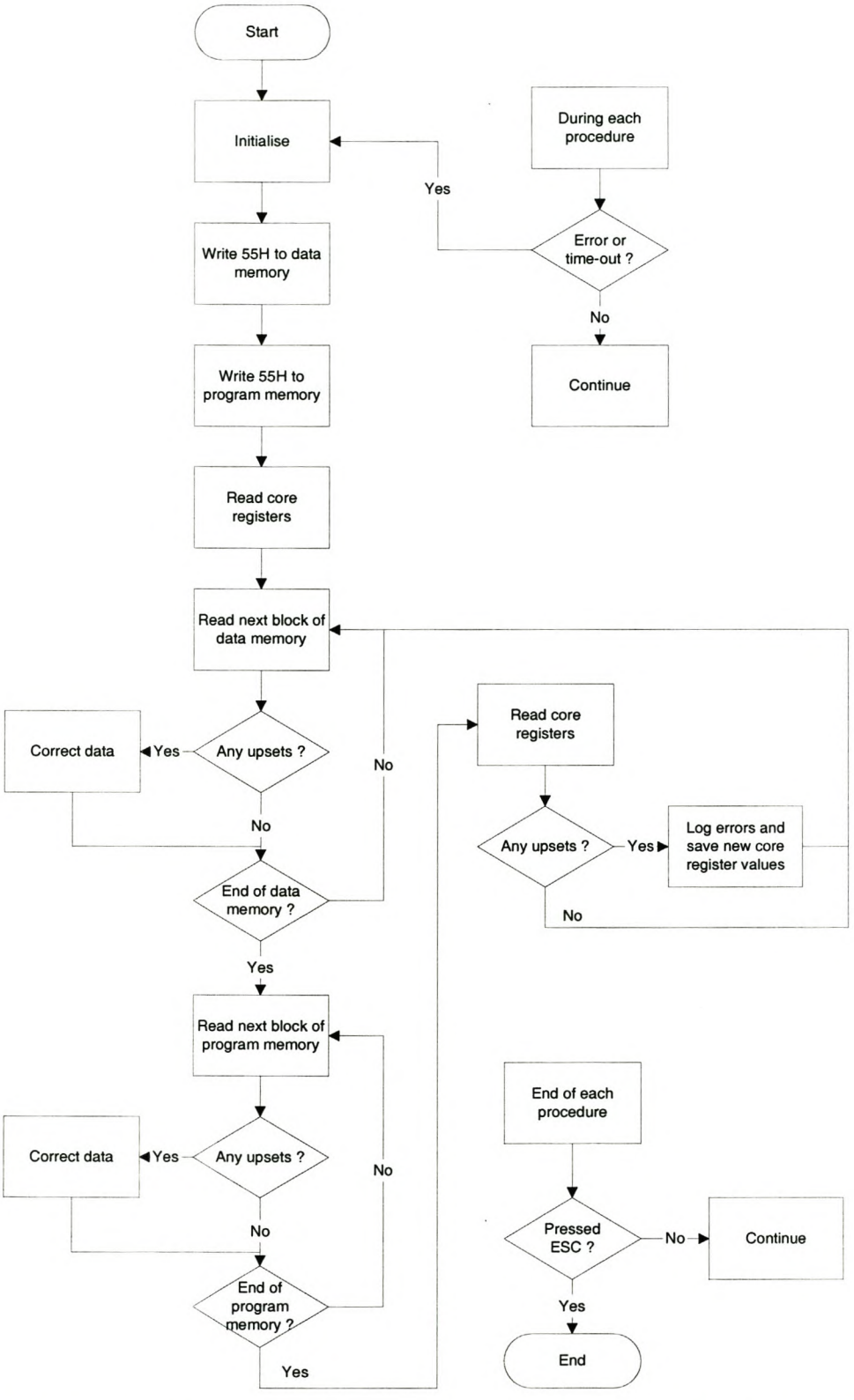



**Figure F.1:** A flow diagram of the memory test program which tests the susceptibility of the DSP processor on-chip memory to upsets.

## F.2 Pascal Code for NOP Test

```
Procedure NOP_test;

{ Does the NOP test }

Var
   Last_clock, Current_clock : Boolean;
   Last_code_pos, Counter,
   Current_port_val          : Byte;
   Parallel_inp_port         : Word;
   Total_upsets              : Word;
   Total_tests               : Longint;
   Hour, Min ,Sec, Sec100    : Word;

Label
   Init_DSP_start;

Begin
   Writeln(' NOP test :');
   Writeln;

   Writeln( Report_file );
   Writeln( Report_file, 'START OF NOP TEST REPORT :' );
   Writeln( Report_file );

   C1 := #0;
   Latchup := False;
   Power_on := False;
   Parallel_inp_port := Parallel_port_adr +1;
   Total_tests := 0;
   Last_clock := False;
   Last_code_pos := 0;

   { Initialise everything for the DSP NOP test }

   Init_DSP_start:                          { Label for start of DSP init }

   Counter := 0;
   Total_upsets := 0;

   { Test for latchup and turn power to DSP on }

   If Latchup = True then
   Begin
      Beep;

      TextColor( Red );
      Writeln(' Latchup occured - turned power off');
      TextColor( LightGray );

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
      Writeln( Report_file, 'Latchup occured - turned power off' );

      Delay( Latchup_delay );

      Nosound;

      Latchup := False;
   End;

   If Power_on = False then
   Begin
      Writeln(' Turning power to DSP on');

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
      Writeln( Report_file, 'Turning power to DSP on' );
                                                   { Turn power on }
```

```
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $02;
   Power_on := True;
End;

{ Manually reset DSP board }

Writeln(' Resetting DSP...');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Resetting DSP...' );

Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $01;
Delay( 1 );
Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FE;

Delay( Boot_delay );                { Wait for board to finish resetting }

{ Measure DSP temperature }

A2D;                                                { Get DSP temperature }

Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C' );

Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
         ' C' );

If Keypressed then
Begin
   C1 := Readkey;
   If C1 = #0 then C2 := Readkey;
end;

{ Download NOP program to DSP }

Writeln(' Downloading NOP program...');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Downloading NOP program to DSP...' );

Exec( 'C:\TESIS\EZ-KIT\EZSHARC\Diag21K.exe',
      '-pC:\TESIS\TOETSP~1\nop2.exe -xscript.txt' );

Writeln;

If DosError <> 0 then                          { Check for execution error }
Begin
   Beep;

   TextColor( Red );
   Writeln(' Dos error #',DosError,' occured !');

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'Dos error #',DosError,' occured !' );

   Writeln(' Rebooting DSP and restarting test');
   Writeln( Report_file, 'Rebooting DSP and restarting test');
   TextColor( LightGray );
                                                   { Turn DSP power off }
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FD;
   Power_on := False;

   Goto Init_DSP_start;
End;

Writeln(' Starting the NOP test');
Writeln;

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Starting the NOP test');
```

```
   Repeat

      If Keypressed then
      Begin
         C1 := Readkey;
         If C1 = #0 then C2 := Readkey;
      End;

      Current_port_val := Port[ Parallel_inp_port ] and $38;
      Current_clock := ( Current_port_val and $08 )=0;

      If Current_clock <> Last_clock then
      Begin
{        Write( Counter,' ');             { Debugging }

         Counter := 0;

         Inc( Total_tests );

         Inc( Last_code_pos );                          { Inc to 3 and wrap }
         Last_code_pos := Last_code_pos and $03;

         If Last_code_pos <> ( Current_port_val shr 4 ) then
         Begin
            Inc( Total_upsets );

            Write('!');

            Last_code_pos := Current_port_val shr 4;
         End;

         Last_clock := Current_clock;
      End
         else
      Begin
         Inc( Counter );

         If Counter = 10 then                           { Test for time-out }
         Begin
            Counter := 0;

            Beep;

            TextColor( Red );
            Writeln(' No response from DSP');
            TextColor( LightBlue );
            Writeln(' Upsets in last run : ', Total_upsets );
            TextColor( LightGray );

            GetTime( Hour, Min ,Sec, Sec100 );
            Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
            Writeln( Report_file, 'No response from DSP');
            Writeln( Report_file, 'Upsets in last run : ', Total_upsets );

            If C1 <> #27 then Goto Init_DSP_start;
         End;

      End;

   until C1 = #27;

   Writeln(' Quitting...');

   TextColor( LightBlue );
   Writeln(' Upsets in last run : ', Total_upsets );
   Writeln(' Total tests done : ', Total_tests );
   TextColor( LightGray );

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'User exit');
   Writeln( Report_file, 'Upsets in last run : ', Total_upsets );
   Writeln( Report_file, 'Total tests done : ', Total_tests );
```

```
    { Measure DSP temperature }

    A2D;                                                { Get DSP temperature }

    Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C'  );

    Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
             ' C' );
                                                        { Turn DSP power off }
    Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FD;
    Power_on := False;

    C1 := #0;

End;   { NOP_test }
```

## F.3 Pascal Code for Cache Test

```
Procedure Cache_test;

{ Does the Cache test }

Var
   Last_clock, Current_clock : Boolean;
   Last_code_pos, Counter,
   Current_port_val          : Byte;
   Parallel_inp_port         : Word;
   Total_upsets              : Word;
   Total_tests               : Longint;
   Hour, Min ,Sec, Sec100    : Word;

Label
   Init_DSP_start;

Begin
   Writeln(' Cache test :');
   Writeln;

   Writeln( Report_file );
   Writeln( Report_file, 'START OF CACHE TEST REPORT :' );
   Writeln( Report_file );

   C1 := #0;
   Latchup := False;
   Power_on := False;
   Parallel_inp_port := Parallel_port_adr +1;
   Total_tests := 0;
   Last_clock := False;
   Last_code_pos := 0;

   { Initialise everything for the DSP NOP test }

   Init_DSP_start:                          { Label for start of DSP init }

   Counter := 0;
   Total_upsets := 0;

   { Test for latchup and turn power to DSP on }

   If Latchup = True then
   Begin
      Beep;

      TextColor( Red );
      Writeln(' Latchup occured - turned power off');
      TextColor( LightGray );

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
      Writeln( Report_file, 'Latchup occured - turned power off' );

      Delay( Latchup_delay );

      Latchup := False;
   End;

   If Power_on = False then
   Begin
      Writeln(' Turning power to DSP on');

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
      Writeln( Report_file, 'Turning power to DSP on' );
                                                    { Turn power on }
      Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $02;
      Power_on := True;
   End;

   { Manually reset DSP board }
```

```
Writeln(' Resetting DSP...');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Resetting DSP...' );

Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $01;
Delay( 1 );
Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FE;

Delay( Boot_delay );                { Wait for board to finish resetting }

{ Measure DSP temperature }

A2D;                                                { Get DSP temperature }
Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C' );

Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
         ' C' );

If Keypressed then
Begin
   C1 := Readkey;
   If C1 = #0 then C2 := Readkey;
End;

{ Download NOP program to DSP }

Writeln(' Downloading Cache program...');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Downloading Cache program to DSP...' );

Exec( 'C:\TESIS\EZ-KIT\EZSHARC\Diag21K.exe',
      '-pC:\TESIS\TOETSP~1\Cache\cache.exe -xscript.txt' );

Writeln;
If DosError <> 0 then                          { Check for execution error }
Begin
   Beep;

   TextColor( Red );
   Writeln(' Dos error #',DosError,' occured !');

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'Dos error #',DosError,' occured !' );

   Writeln(' Rebooting DSP and restarting test');
   Writeln( Report_file, 'Rebooting DSP and restarting test');
   TextColor( LightGray );
                                                   { Turn DSP power off }
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FD;
   Power_on := False;

   Goto Init_DSP_start;
End;

Writeln(' Starting the Cache test');
Writeln;

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Starting the Cache test');

Repeat

   If Keypressed then
   Begin
      C1 := Readkey;
      If C1 = #0 then C2 := Readkey;
   End;
```

```
      Current_port_val := Port[ Parallel_inp_port ] and $38;
      Current_clock := ( Current_port_val and $08 )=0;

      If Current_clock <> Last_clock then
      Begin
{         Write( Counter,' ');              { Debugging }

         Counter := 0;

         Inc( Total_tests );

         Inc( Last_code_pos );                          { Inc to 3 and wrap }
         Last_code_pos := Last_code_pos and $03;

         If Last_code_pos <> ( Current_port_val shr 4 ) then
         Begin
            Inc( Total_upsets );

            Write('!');

            Last_code_pos := Current_port_val shr 4;
         End;

         Last_clock := Current_clock;
      End
         else
      Begin
         Inc( Counter );

         If Counter = 10 then                           { Test for time-out }
         Begin
            Counter := 0;

            Beep;

            TextColor( Red );
            Writeln(' No response from DSP');
            TextColor( LightBlue );
            Writeln(' Upsets in last run : ', Total_upsets );
            TextColor( LightGray );

            GetTime( Hour, Min ,Sec, Sec100 );
            Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
            Writeln( Report_file, 'No response from DSP');
            Writeln( Report_file, 'Upsets in last run : ', Total_upsets );

            If C1 <> #27 then Goto Init_DSP_start;
         End;

      End;

   until C1 = #27;

   Writeln(' Quitting...');

   TextColor( LightBlue );
   Writeln(' Upsets in last run : ', Total_upsets );
   Writeln(' Total tests done : ', Total_tests );
   TextColor( LightGray );

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'User exit');
   Writeln( Report_file, 'Upsets in last run : ', Total_upsets );
   Writeln( Report_file, 'Total tests done : ', Total_tests );

   { Measure DSP temperature }

   A2D;                                             { Get DSP temperature }

   Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
            ' C' );
   Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C'  );
```

```
                                                    { Turn DSP power off }
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FD;
   Power_on := False;

   C1 := #0;

End;   { Cache_test }
```

## F.4 Pascal Code for ALU Test

```
Procedure ALU_test;

{ Does the ALU test }

Var

   Current_clock, Last_clock      : Boolean;
   Counter, Current_port_val      : Byte;
   Parallel_inp_port,
   Logic_upsets, Math_upsets,
   Total_tests                    : Longint;
   Hour, Min ,Sec, Sec100         : Word;

Label
   Init_DSP_start;

Begin
   Writeln(' ALU test :');
   Writeln;

   Writeln( Report_file );
   Writeln( Report_file, 'START OF ALU TEST REPORT :' );
   Writeln( Report_file );

   C1 := #0;
   Latchup := False;
   Power_on := False;
   Parallel_inp_port := Parallel_port_adr +1;
   Total_tests := 0;

   { Initialise everything for the DSP ALU test }

   Init_DSP_start:                          { Label for start of DSP init }

   Counter := 0;
   Logic_upsets := 0;
   Math_upsets := 0;

   { Test for latchup and turn power to DSP on }

   If Latchup = True then
   Begin
      Beep;

      TextColor( Red );
      Writeln(' Latchup occured - turned power off');
      TextColor( LightGray );

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
      Writeln( Report_file, 'Latchup occured - turned power off' );

      Delay( Latchup_delay );
      Nosound;
      Latchup := False;
   End;

   If Power_on = False then
   Begin
      Writeln(' Turning power to DSP on');

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
      Writeln( Report_file, 'Turning power to DSP on' );
                                                        { Turn power on }
      Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $02;
      Power_on := True;
   End;
```

```
{ Manually reset DSP board }

Writeln(' Resetting DSP...');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Resetting DSP...' );

Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $01;
Delay( 1 );
Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FE;

Delay( Boot_delay );                { Wait for board to finish resetting }

{ Measure DSP temperature }

A2D;                                                { Get DSP temperature }
Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
         ' C' );
Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C' );

If Keypressed then
Begin
   C1 := Readkey;
   If C1 = #0 then C2 := Readkey;
end;

{ Download ALU program to DSP }

Writeln(' Downloading ALU program...');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Downloading ALU program to DSP...' );

Exec( 'C:\TESIS\EZ-KIT\EZSHARC\Diag21K.exe',
      '-pC:\TESIS\TOETSP~1\ALU\alu2.exe -xscript.txt' );

Writeln;

If DosError <> 0 then                          { Check for execution error }
Begin
   Beep;

   TextColor( Red );
   Writeln(' Dos error #',DosError,' occured !');

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'Dos error #',DosError,' occured !' );

   Writeln(' Rebooting DSP and restarting test');
   Writeln( Report_file, 'Rebooting DSP and restarting test');
   TextColor( LightGray );
                                                    { Turn DSP power off }
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FD;
   Power_on := False;

   Goto Init_DSP_start;
End;

Writeln(' Starting the ALU test');
Writeln;

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Starting the ALU test');

Last_clock := False;

Repeat
   If Keypressed then
   Begin
```

```
        C1 := Readkey;
        If C1 = #0 then C2 := Readkey;
      end;

      Current_port_val := Port[ Parallel_inp_port ];
      Current_clock := ( Current_port_val and $08 )=0;

      If Current_clock <> Last_clock then
      Begin
{         Write( Counter,' ');          { Debugging }

        Counter := 0;
                                              { Add 1 to upset if bit set }
        Inc( Total_tests );

        Inc( Logic_upsets, ( Current_port_val shr 4 ) and $01 );   { Flag 2 }
        Inc( Math_upsets, ( Current_port_val shr 5 ) and $01 );    { Flag 3 }

        Last_clock := Current_clock;
      End
        else
      Begin
        Inc( Counter );

        If Counter = 10 then                          { Test for time-out }
        Begin
          Counter := 0;

          Beep;

          TextColor( Red );
          Writeln(' No response from DSP');
          TextColor( LightBlue );
          Writeln(' Logic upsets in last run : ', Logic_upsets );
          Writeln(' Math upsets in last run : ', Math_upsets );
          TextColor( LightGray );

          GetTime( Hour, Min ,Sec, Sec100 );
          Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
          Writeln( Report_file, 'No response from DSP');

          Writeln( Report_file, 'Logic upsets in last run : ', Logic_upsets );
          Writeln( Report_file, 'Math upsets in last run : ', Math_upsets );

          If C1 <> #27 then Goto Init_DSP_start;
        End;

      End;

  until C1 = #27;

  Writeln(' Quitting...');

  TextColor( LightBlue );
  Writeln(' Logic upsets in last run : ', Logic_upsets );
  Writeln(' Math upsets in last run : ', Math_upsets );
  Writeln(' Total tests done : ', Total_tests );
  TextColor( LightGray );

  GetTime( Hour, Min ,Sec, Sec100 );
  Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
  Writeln( Report_file, 'User signalled exit');

  Writeln( Report_file, 'Logic upsets in last run : ', Logic_upsets );
  Writeln( Report_file, 'Math upsets in last run : ', Math_upsets );
  Writeln( Report_file, 'Total tests done : ', Total_tests );

  { Measure DSP temperature }

  A2D;                                          { Get DSP temperature }

  Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C' );

  Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
```

```
            ' C' );
                                                    { Turn DSP power off }
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FD;
   Power_on := False;

   C1 := #0;

End;   { ALU_test }
```

## F.5 Pascal Code for Memory Test

```
Procedure Memory_test;

{ Does the memory test }

Var
   Upset                        : Boolean;              { Set if upset occured }
   Retry_command                : Boolean;     { Set if sending command again }

   Total_upsets                 : Word;        { Total number of upsets found }

   Mem_adr_low, Mem_adr_high    : Word;           { The current adr in memory }

   Error                        : Byte;        { Error returned by Get_packet }
   Data                         : Byte;          { Data read with memory read }
   Counter1, Counter2           : Word;             { General purpose counter }
   Total_tests                  : Longint;

   Core_registers               : Array[ 0..55 ] of Byte;  { Stores registers }

   Hour, Min, Sec, Sec100       : Word;

   I                            : Word;                     { General counter }

Label
   Init_DSP_start,
   Test_end,
   Serial_speed_set,
   Verify_comms_start,
   Write_DM_startup,
   Write_PM_startup,
   Read_DM, Read_PM,
   Read_core_start,
   Core_test_start,
   Correct_DM_upset,
   Correct_PM_upset;

Begin
   Writeln(' Memory test :');
   Writeln;

   Writeln( Report_file );
   Writeln( Report_file, 'START OF MEMORY TEST REPORT :' );
   Writeln( Report_file );

   Retry_command := False;

   Mem_adr_low := DM_start_adr;                        { First memory address }
   Mem_adr_high := $0002;

   C1 := #0;
   Total_upsets := 0;
   Latchup := False;
   Power_on := False;
   Total_tests := 0;

   { Initialise everything for the DSP memory test }

   Init_DSP_start:                               { Label for start of DSP init }

   If Keypressed then
   Begin
      C1 := Readkey;
      If C1 = #0 then C2 := Readkey;
   End;

   { Test for latchup and turn power to DSP on }

   If Latchup = True then
```

```
Begin
   Beep;

   TextColor( Red );
   Writeln(' Latchup occured - turned power off');
   TextColor( LightGray );

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'Latchup occured - turned power off' );

   Delay( Latchup_delay );

   Nosound;

   Latchup := False;
End;

If Power_on = False then
Begin
   Writeln(' Turning power to DSP on');

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'Turning power to DSP on' );
                                                        { Turn power on }
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $02;
   Power_on := True;
End;


Serial_init( 9600, $03 );                    { Init computer serial port }

{ Manually reset DSP board }

Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] or $01;
Delay( 1 );
Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FE;

Delay( Boot_delay );                { Wait for board to finish resetting }

{ Synchronize comms }

Send_serial_data( @Synch_comms );

If C1 = #27 then Goto Test_end;                          { Check for quit }


{ Reset processor }

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Resetting processor' );

Send_serial_data( @Reset_processor );

Error := Get_packet;
If Error <> 0 then
Begin
   Beep;

   TextColor( Red );
   Writeln(' Could not reset processor');
   Writeln( Error_string( Error ) );
   TextColor( LightGray );

   Writeln( Report_file, 'Could not reset processor');
   Writeln( Report_file, Error_string( Error ) );

   Clear_buf;

   Writeln( Report_file, 'Trying again...');
   Writeln(' Trying again...');
```

```
   Goto Init_DSP_start;

End;

Delay( Boot_delay );                          { Wait to finish resetting }

Send_serial_data( @Synch_comms );               { Synch comms with board }


{ Init the DSP board to Serial_speed baud }

Serial_speed_set:                  { Label }

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

Writeln( Report_file, 'Setting serial speed to ', Serial_speed, ' baud' );

Send_serial_data( @Set_serial_speed );    { Set DSP speed to Serial_speed }
Send_serial_byte( ( 1152000 div Serial_speed ) mod 256 );
Send_serial_byte( 0 );
Send_serial_byte( 0 );
Send_serial_byte( 0 );
Send_serial_byte( ( 1152000 div Serial_speed ) div 256  );
Send_serial_byte( 0 );
Send_serial_byte( 0 );
Send_serial_byte( 0 );
                                                  { Wait until all sent }
Repeat until ( Port[ Serial_port_adr +LSR ] and $60 ) = $60;

Serial_init( Serial_speed, $03 );                   { Set computer speed }

Delay( Set_speed_delay );                     { Wait for DSP to set speed }

Error := Get_packet;
If Error <> 0 then
Begin
   Beep;


   TextColor( Red );
   Writeln(' Could not set serial speed');
   Writeln( Error_string( Error ) );
   TextColor( LightGray );

   Writeln( Report_file, 'Could not set serial speed');
   Writeln( Report_file, Error_string( Error ) );

   Clear_buf;

   Serial_init( 9600, $03 );                { Set computer speed to 9,6k }

   If ( Retry_command = True ) or ( Latchup = True ) then
   Begin
      Retry_command := False;

      Writeln(' Rebooting DSP and restarting test');
      Writeln( Report_file, 'Rebooting DSP and restarting test');

      Goto Init_DSP_start;
   End
      else
   Begin
      Retry_command := True;

      Writeln(' Trying again...');
      Writeln( Report_file, 'Trying again...');

      Goto Serial_speed_set;
   End;

End;

If C1 = #27 then Goto Test_end;                          { Check for quit }
```

```
{ Do verify comms to test for faults }

Verify_comms_start :                                          { Label }

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

Writeln( Report_file, 'Verifying communication with board');

Send_serial_data( @Verify_comms );

Error := Get_packet;
If Error <> 0 then
Begin
   Beep;

   TextColor( Red );
   Writeln(' Verify comms failed');
   Writeln( Error_string( Error ) );
   TextColor( LightGray );

   Writeln( Report_file, ' Verify comms failed');
   Writeln( Report_file, Error_string( Error ) );

   Clear_buf;

   If ( Retry_command = True ) or ( Latchup = True ) then
   Begin
      Retry_command := False;

      Writeln(' Rebooting DSP and restarting test');
      Writeln( Report_file, 'Rebooting DSP and restarting test');

      Goto Init_DSP_start;
   End
      else
   Begin
      Retry_command := True;

      Writeln(' Trying again...');
      Writeln( Report_file, 'Trying again...');

      Goto Serial_speed_set;
   End;

End;

{ Measure DSP temperature }

A2D;                                          { Get DSP temperature }

Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C'  );

Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
         ' C' );

{ Write test byte to all DM locations }

Writeln(' Starting to write $',Byte2Hex( Mem_test_byte ),' to DM');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Starting to write $',Byte2Hex( Mem_test_byte ),' to DM');

Mem_adr_low := DM_start_adr;
Mem_adr_high := $0002;

Repeat

   Write_DM_startup :                                         { Label }

   If Keypressed then
   Begin
```

```
      C1 := Readkey;
      If C1 = #0 then C2 := readkey;
   End;

   Send_serial_data( @Write_DM32 );

   Send_serial_byte( Lo( Mem_adr_low ) );              { Send start adr }
   Send_serial_byte( Hi( Mem_adr_low ) );
   Send_serial_byte( Lo( Mem_adr_high ) );
   Send_serial_byte( Hi( Mem_adr_high ) );

   Send_serial_byte( DM_block_size );                  { Send block size }
   Send_serial_byte( 0 );
   Send_serial_byte( 0 );
   Send_serial_byte( 0 );

   For Counter1 := 1 to DM_block_size shl 2 do      { Send data to write }
      Send_serial_byte( Mem_test_byte );

   WriteXY( 65, 10, Mem_adr_low );

   Error := Get_packet;
   If Error <> 0 then
   Begin
      Beep;

      TextColor( Red );
      Writeln(' Could not write to DM location : ', Mem_adr_low );
      Writeln( Error_string( Error ) );
      TextColor( LightGray );

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

      Writeln( Report_file, 'Could not write to DM location : ',
               Mem_adr_low );
      Writeln( Report_file, Error_string( Error ) );

      Clear_buf;                                    { Clear serial buffer }

      If ( Retry_command = True ) or ( Latchup = True ) then
      Begin
         Retry_command := False;

         Writeln(' Rebooting DSP and restarting test');
         Writeln( Report_file, 'Rebooting DSP and restarting test');

         Goto Init_DSP_start;
      End
         else
      Begin
         Retry_command := True;

         Writeln(' Trying again...');
         Writeln( Report_file, 'Trying again...');

         Goto Write_DM_startup;
      End;

   end;

   Inc( Mem_adr_low, DM_block_size );

until ( Mem_adr_low > DM_end_adr ) or ( C1 = #27 );

If C1 = #27 then Goto Test_end;

Writeln(' Finished writing $',Byte2Hex( Mem_test_byte ),' to DM');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Finished writing $',Byte2Hex( Mem_test_byte ),' to DM');
```

```
{ Write test byte to PM locations }

Writeln(' Starting to write $',Byte2Hex( Mem_test_byte ),' to PM');

Writeln( Report_file, 'Starting to write $',Byte2Hex( Mem_test_byte ),' to PM');

Mem_adr_low := PM_start_adr;
Mem_adr_high := $0002;

Repeat
   Write_PM_startup :                                          { Label }

   If Keypressed then
   Begin
      C1 := Readkey;
      If C1 = #0 then C2 := readkey;
   End;

   Send_serial_data( @Write_PM48 );

   Send_serial_byte( Lo( Mem_adr_low ) );                { Send start adr }
   Send_serial_byte( Hi( Mem_adr_low ) );
   Send_serial_byte( Lo( Mem_adr_high ) );
   Send_serial_byte( Hi( Mem_adr_high ) );

   Send_serial_byte( PM_block_size );                   { Send block size }
   Send_serial_byte( 0 );
   Send_serial_byte( 0 );
   Send_serial_byte( 0 );

   For Counter1 := 1 to PM_block_size do             { Send data to write }
   Begin
      For Counter2 := 1 to 6 do Send_serial_byte( Mem_test_byte );
      Send_serial_byte( $00 );
      Send_serial_byte( $00 );
   End;

   WriteXY( 65, 10, Mem_adr_low );

   Error := Get_packet;
   If Error <> 0 then
   Begin
      Beep;

      TextColor( Red );
      Writeln(' Could not write to PM location : ', Mem_adr_low );
      Writeln( Error_string( Error ) );
      TextColor( LightGray );

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

      Writeln( Report_file, 'Could not write to PM location : ',
               Mem_adr_low );
      Writeln( Report_file, Error_string( Error ) );

      Clear_buf;                                     { Clear serial buffer }

      If ( Retry_command = True ) or ( Latchup = True ) then
      Begin
         Retry_command := False;

         Writeln(' Rebooting DSP and restarting test');
         Writeln( Report_file, 'Rebooting DSP and restarting test');

         Goto Init_DSP_start;
      End
         else
      Begin
         Retry_command := True;

         Writeln(' Trying again...');
         Writeln( Report_file, 'Trying again...');
```

```
         Goto Write_PM_startup;
      End;

   end;

   Inc( Mem_adr_low, PM_block_size );

until ( Mem_adr_low > PM_end_adr ) or ( C1 = #27 );

If C1 = #27 then Goto Test_end;

Writeln(' Finished writing $',Byte2Hex( Mem_test_byte ),' to PM');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
Writeln( Report_file, 'Finished writing $',Byte2Hex( Mem_test_byte ),' to PM');


{ Read core registers and store }

Read_core_start:

Writeln(' Reading core registers');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

Writeln( Report_file, 'Reading core registers');

Send_serial_data( @Read_core );

Error := Get_packet;
If Error <> 0 then
Begin
   Beep;

   TextColor( Red );
   Writeln(' Could not read core registers' );
   Writeln( Error_string( Error ) );
   TextColor( LightGray );

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

   Writeln( Report_file, 'Could not read core registers' );
   Writeln( Report_file, Error_string( Error ) );

   Clear_buf;                                  { Clear serial buffer }

   If ( Retry_command = True ) or ( Latchup = True ) then
   Begin
      Retry_command := False;

      Writeln(' Rebooting DSP and restarting test');
      Writeln( Report_file, 'Rebooting DSP and restarting test');

      Goto Init_DSP_start;
   End
      else
   Begin
      Retry_command := True;

      Writeln(' Trying again...');
      Writeln( Report_file, 'Trying again...');

      Goto Read_core_start;
   End;

End;

For I := 0 to 55 do                          { Store registers in variable }
   Core_registers[ I ] := Get_serial_byte;
```

```
Writeln(' Starting the Memory test');

{ Start with the DSP memory test }

Repeat

   If Keypressed then
   Begin
      C1 := Readkey;
      If C1=#0 then C2 := Readkey;
   End;

   { Start with Data Memory test }

   Writeln(' Starting DM test cycle');

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

   Writeln( Report_file, 'Starting DM test cycle');

   Mem_adr_low := DM_start_adr;
   Mem_adr_high := $0002;

   Repeat

      { Read block of memory }

      Read_DM :                                                { Label }

      If Keypressed then
      Begin
         C1 := Readkey;
         If C1 = #0 then C2 := readkey;
      End;

      Send_serial_data( @Read_DM32 );

      Send_serial_byte( Lo( Mem_adr_low ) );             { Send start adr }
      Send_serial_byte( Hi( Mem_adr_low ) );
      Send_serial_byte( Lo( Mem_adr_high ) );
      Send_serial_byte( Hi( Mem_adr_high ) );

      Send_serial_byte( DM_block_size );                { Send block size }
      Send_serial_byte( 0 );
      Send_serial_byte( 0 );
      Send_serial_byte( 0 );

      Error := Get_packet;
      If Error <> 0 then
      Begin
         Beep;

         TextColor( Red );
         Writeln(' Could not read block of DM : ', Mem_adr_low );
         Writeln( Error_string( Error ) );
         TextColor( LightGray );

         GetTime( Hour, Min ,Sec, Sec100 );
         Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

         Writeln( Report_file, 'Could not read block of DM : ',
                  Mem_adr_low );
         Writeln( Report_file, Error_string( Error ) );

         Clear_buf;                                  { Clear serial buffer }

         If ( Retry_command = True ) or ( Latchup = True ) then
         Begin
            Retry_command := False;

            Writeln(' Rebooting DSP and restarting test');
            Writeln( Report_file, 'Rebooting DSP and restarting test');
```

```
      Goto Init_DSP_start;
   End
      else
   Begin
      Retry_command := True;

      Writeln(' Trying again...');
      Writeln( Report_file, 'Trying again...');

      Goto Read_DM;
   End;

End;

Inc( Total_tests, DM_Block_size );

{ Test for any single event upsets }

Upset := False;

For Counter1 := 1 to DM_Block_size shl 2 do
Begin

   Data := Get_serial_byte;

   If Data <> Mem_test_byte then
   Begin

      Upset := True;
      Inc( Total_upsets );

      TextColor( Red );
      Writeln(' Upset byte = ', Data );

      Writeln(' Upset #',Total_upsets,' in adr block ',
               Mem_adr_low, ' position # ', Counter1 );
      TextColor( LightGray );

      GetTime( Hour, Min ,Sec, Sec100 );
      Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

      Writeln( Report_file, 'Upset byte = ', Data );

      Writeln( Report_file, 'Upset #',Total_upsets,' in adr block ',
               Mem_adr_low, ' position # ', Counter1 );
   End;

End;   { For }

If Upset = True then                          { If upset, correct }
Begin
   Correct_DM_upset :                                    { Label }

   Writeln(' Correcting upset');

   Writeln( Report_file, 'Correcting upset');

   Send_serial_data( @Write_DM32 );

   Send_serial_byte( Lo( Mem_adr_low ) );          { Send start adr }
   Send_serial_byte( Hi( Mem_adr_low ) );
   Send_serial_byte( Lo( Mem_adr_high ) );
   Send_serial_byte( Hi( Mem_adr_high ) );

   Send_serial_byte( DM_block_size );             { Send block size }
   Send_serial_byte( 0 );
   Send_serial_byte( 0 );
   Send_serial_byte( 0 );

   For Counter1 := 1 to DM_block_size shl 2 do { Send data to write }
      Send_serial_byte( Mem_test_byte );

   Error := Get_packet;                          { Check for errors }
   If Error <> 0 then
```

```
      Begin
         Beep;

         TextColor( Red );
         Writeln(' Could not correct upset byte(s)');
         Writeln( Error_string( Error ) );
         TextColor( LightGray );

         Writeln( Report_file, 'Could not correct upset byte(s)');
         Writeln( Report_file, Error_string( Error ) );

         Clear_buf;                              { Clear serial buffer }

         If ( Retry_command = True ) or ( Latchup = True ) then
         Begin
            Retry_command := False;

            Writeln(' Rebooting DSP and restarting test');
            Writeln( Report_file, 'Rebooting DSP and restarting test');

            Goto Init_DSP_start;
         End
            else
         Begin
            Retry_command := True;

            Writeln(' Trying again...');
            Writeln( Report_file, 'Trying again...');

            Goto Correct_DM_upset;
         End;

      End;

   End;   { If }

   WriteXY( 65, 12, Mem_adr_low );

   Inc( Mem_adr_low, DM_block_size );

until ( C1 = #27 ) or ( Mem_adr_low > DM_end_adr );

If C1 = #27 then Goto Test_end;

Writeln(' End of DM test cycle');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

Writeln( Report_file, 'End of DM test cycle');


{ Start with PM test }

Writeln(' Starting with PM test cycle');

Writeln( Report_file, 'Starting PM test cycle' );

Mem_adr_low := PM_start_adr;
Mem_adr_high := $0002;

Repeat
   { Read block of memory }

   Read_PM :                                              { Label }

   If Keypressed then
   Begin
      C1 := Readkey;
      If C1 = #0 then C2 := Readkey;
   End;

   Send_serial_data( @Read_PM48 );
```

```
Send_serial_byte( Lo( Mem_adr_low ) );              { Send start adr }
Send_serial_byte( Hi( Mem_adr_low ) );
Send_serial_byte( Lo( Mem_adr_high ) );
Send_serial_byte( Hi( Mem_adr_high ) );

Send_serial_byte( PM_block_size );                  { Send block size }
Send_serial_byte( 0 );
Send_serial_byte( 0 );
Send_serial_byte( 0 );

Error := Get_packet;
If Error <> 0 then
Begin
   Beep;

   TextColor( Red );
   Writeln(' Could not read block of PM : ', Mem_adr_low );
   Writeln( Error_string( Error ) );
   TextColor( LightGray );

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

   Writeln( Report_file, 'Could not read block of PM : ',
            Mem_adr_low );
   Writeln( Report_file, Error_string( Error ) );

   Clear_buf;                                    { Clear serial buffer }

   If ( Retry_command = True ) or ( Latchup = True ) then
   Begin
      Retry_command := False;

      Writeln(' Rebooting DSP and restarting test');
      Writeln( Report_file, 'Rebooting DSP and restarting test');

      Goto Init_DSP_start;
   End
      else
   Begin
      Retry_command := True;

      Writeln(' Trying again...');
      Writeln( Report_file, 'Trying again...');

      Goto Read_PM;
   End;

End;

Inc( Total_tests, Trunc( PM_Block_size * 1.5 ) );   { PM is 48 bits }

{ Test for any single event upsets }

Upset := False;

For Counter1 := 1 to PM_Block_size do
Begin
   For Counter2 := 1 to 8 do
   Begin

      Data := Get_serial_byte;
                                          { Only test first 6 bytes }
      If ( Data <> Mem_test_byte ) and ( Counter2 < 7 ) then
      Begin
         TextColor( Red );
         Writeln(' Upset byte = ', Data );

         Writeln(' Upset #',Total_upsets,' in adr block ',
                  Mem_adr_low, ' position # ', Counter1 );

         TextColor( LightGray );

         GetTime( Hour, Min ,Sec, Sec100 );
```

```
            Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

            Writeln( Report_file, 'Upset byte = ', Data );

            Upset := True;
            Inc( Total_upsets );

            Writeln( Report_file, 'Upset #',Total_upsets,' in adr block ',
                     Mem_adr_low, ' position # ', Counter1 );
         End;

      End;   { For }

   End;   { For }

   If Upset = True then                              { If upset, correct }
   Begin
      Correct_PM_upset :                                        { Label }

      Writeln(' Correcting upset');

      Writeln( Report_file, 'Correcting upset');

      Send_serial_data( @Write_PM48 );

      Send_serial_byte( Lo( Mem_adr_low ) );          { Send start adr }
      Send_serial_byte( Hi( Mem_adr_low ) );
      Send_serial_byte( Lo( Mem_adr_high ) );
      Send_serial_byte( Hi( Mem_adr_high ) );

      Send_serial_byte( PM_block_size );             { Send block size }
      Send_serial_byte( 0 );
      Send_serial_byte( 0 );
      Send_serial_byte( 0 );

      For Counter1 := 1 to PM_block_size do       { Send data to write }
      Begin
         For Counter2 := 1 to 6 do Send_serial_byte( Mem_test_byte );
         Send_serial_byte( $00 );
         Send_serial_byte( $00 );
      End;

      Error := Get_packet;                           { Check for errors }
      If Error <> 0 then
      Begin
         Beep;

         TextColor( Red );
         Writeln(' Could not correct upset byte(s)');
         Writeln( Error_string( Error ) );
         TextColor( LightGray );

         Writeln( Report_file, 'Could not correct upset byte(s)');
         Writeln( Report_file, Error_string( Error ) );

         Clear_buf;                                 { Clear serial buffer }

         If ( Retry_command = True ) or ( Latchup = True ) then
         Begin
            Retry_command := False;

            Writeln(' Rebooting DSP and restarting test');
            Writeln( Report_file, 'Rebooting DSP and restarting test');

            Goto Init_DSP_start;
         End
            else
         Begin
            Retry_command := True;

            Writeln(' Trying again...');
            Writeln( Report_file, 'Trying again...');

            Goto Correct_PM_upset;
```

```
         End;

      End;

   End;   { If }

   WriteXY( 65, 12, Mem_adr_low );

   Inc( Mem_adr_low, PM_block_size );

until ( C1 = #27 ) or ( Mem_adr_low > PM_end_adr );

If C1 = #27 then Goto Test_end;

Writeln(' End of PM test cycle');

GetTime( Hour, Min ,Sec, Sec100 );
Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

Writeln( Report_file, 'End of PM test cycle');


{ Start with core register test }

Core_test_start:

If Keypressed then
Begin
   C1 := Readkey;
   If C1 = #0 then C2 := readkey;
End;

Writeln(' Starting core register test');

Writeln( Report_file, 'Starting core register test');

Send_serial_data( @Read_core );

Error := Get_packet;
If Error <> 0 then
Begin
   Beep;

   TextColor( Red );
   Writeln(' Could not read core registers' );
   Writeln( Error_string( Error ) );
   TextColor( LightGray );

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );

   Writeln( Report_file, 'Could not read core registers' );
   Writeln( Report_file, Error_string( Error ) );

   Clear_buf;                                  { Clear serial buffer }

   If ( Retry_command = True ) or ( Latchup = True ) then
   Begin
      Retry_command := False;

      Writeln(' Rebooting DSP and restarting test');
      Writeln( Report_file, 'Rebooting DSP and restarting test');

      Goto Init_DSP_start;
   End
      else
   Begin
      Retry_command := True;

      Writeln(' Trying again...');
      Writeln( Report_file, 'Trying again...');

      Goto Core_test_start;
   End;
```

```
      End;

      Inc( Total_tests, 9 );                              { 9 registers read }

      Upset := False;

      For I := 0 to 55 do           { Compare registers with previous values }
      Begin
         Data := Get_serial_byte;
         If Core_registers[ I ] <> Data then
         Begin
            Upset := True;
            Inc( Total_upsets );
            Core_registers[ I ] := Data;    { Updata variable with new value }
         End;
      End;

      If Upset = True then
      Begin
         Writeln(' Core register(s) changed');
         Writeln( Report_file, 'Core register(s) changed');
      End;

      Writeln(' Core register test finished');
      Writeln( Report_file, 'Core register test finished');

   until C1 = #27;


   Test_end:

   Writeln(' Quitting...');

   TextColor( LightBlue );
   Writeln(' Tested ', Total_tests, ' memory blocks of 32 bits each');
   Writeln(' Found ', Total_upsets, ' upsets in the memory');
   TextColor( LightGray );

   GetTime( Hour, Min ,Sec, Sec100 );
   Writeln( Report_file, '[', Hour, ':', Min, ':', Sec, ']' );
   Writeln( Report_file, 'User signalled exit');
   Writeln( Report_file, 'Tested ', Total_tests, ' memory blocks of 32 bits each');
   Writeln( Report_file, 'Found ', Total_upsets, ' upsets in the memory');

   { Measure DSP temperature }

   A2D;                                              { Get DSP temperature }
   Writeln( Report_file,'DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0,
            ' C' );
   Writeln(' DSP Temperature : ', ( A2D_data[0]*0.48828 -273 ) :3:0, ' C'  );

   Serial_init( 9600, $03 );             { Set computer serial speed to 9600 }

                                                     { Turn DSP power off }
   Port[ Parallel_port_adr ] := Port[ Parallel_port_adr ] and $FD;
   Power_on := False;

{   Serial_restore; }

   C1 := #0;

End;   { Memory_test }
```

# Appendix G

# Code Listing and Simulation for EDAC Circuit

The VHDL code listing and a simulation of the EDAC circuit operation is given in this appendix.

```
--
-- An EDAC implementation in a FPGA to detect/correct upsets in a DSP's
-- program memory
--
-- 20-06-2001
--

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_unsigned.all;


entity EDAC is

   port( data_port : inout std_logic_vector ( 15 downto 0 );  -- Data port
         addr_port : inout std_logic_vector ( 31 downto 0 );  -- Address port

         nHBR : out std_logic;                          -- Host Bus Request
         nHBG : in std_logic;                           -- Host Bus Grant
         nCS : out std_logic;                           -- Chip Select
         REDY : in std_logic;                           -- Host Bus Acknowledge
         nSBTS : out std_logic;                         -- Suspend Bus Tristate

         nRD_port : inout std_logic;                    -- memory Read strobe
         nWR_port : inout std_logic;                    -- memory Write strobe
         nSW : out std_logic;                           -- Synchronous Write select
         PAGE : out std_logic;                          -- dram Page boundary

         nMS : out std_logic_vector ( 3 downto 0 );     -- Memory Select
         ADRCLK : out std_logic;                        -- Clock output reference
         nDMAG1 : out std_logic;                        -- DMA Grant 1
         nDMAG2 : out std_logic;                        -- DMA Grant 2

         clk : in std_logic;                            -- clock for states

         Reset : in std_logic;

end EDAC;

architecture behavioral of EDAC is


   type Main_states is ( Wait_start, Read_PM, Generate_ham, Write_PM );
   type Read_states is ( Request_HB, Wait_HBG, Wait_REDY, Store_words,
                         Release_HB );
   type Generate_ham_states is ( Calculate_ham, Check_data, Check_Hamming );
   type Write_states is ( Request_HB, Write_words, Wait_HBG, Wait_REDY,
                          Release_HB );

   type Data_types is ( Data, Hamming );

----------------------

   signal Addr_out : std_logic_vector ( 31 downto 0 );    -- Connection with port
   signal Addr_in : std_logic_vector ( 31 downto 0 );     -- Connection with port
   signal Addr_count : std_logic_vector ( 15 downto 0 );  -- Address counter
   signal Next_PM_addr : std_logic_vector( 31 downto 0 );
   signal Next_Hamming_addr : std_logic_vector( 31 downto 0 );

   signal Addr_dif : std_logic_vector ( 15 downto 0 );    -- To make Altera compile !

   signal Data_in : std_logic_vector ( 15 downto 0 );     -- Connection with port
   signal Data_out : std_logic_vector ( 15 downto 0 );    -- Connection with port

   signal PM_read : std_logic_vector ( 47 downto 0 );       -- Whole 48 bits read
   signal PM_write : std_logic_vector ( 47 downto 0 );      -- Whole 48 bits to write

   signal Hamming_read : std_logic_vector ( 47 downto 0 );  -- Whole 48 bits read
   signal Hamming_write : std_logic_vector ( 47 downto 0 ); -- Whole 48 bits to write

   signal Ham_new0 : std_logic_vector ( 5 downto 0 );
   signal Ham_new1 : std_logic_vector ( 5 downto 0 );
   signal Ham_new2 : std_logic_vector ( 5 downto 0 );
```

```
   signal Ham_syndrome0 : std_logic_vector ( 5 downto 0 );
   signal Ham_syndrome1 : std_logic_vector ( 5 downto 0 );
   signal Ham_syndrome2 : std_logic_vector ( 5 downto 0 );

   signal Ham_decode0 : std_logic_vector ( 6 downto 0 );
   signal Ham_decode1 : std_logic_vector ( 6 downto 0 );
   signal Ham_decode2 : std_logic_vector ( 6 downto 0 );

   signal Data_error0 : std_logic;         -- Indicates an error in the data
   signal Data_error1 : std_logic;         -- Indicates an error in the data
   signal Data_error2 : std_logic;         -- Indicates an error in the data
   signal Update_data : std_logic;        -- Indicates data has been corrected

   signal Hamming_error0 : std_logic;      -- Indicates an error in the Hamming code
   signal Hamming_error1 : std_logic;      -- Indicates an error in the Hamming code
   signal Hamming_error2 : std_logic;      -- Indicates an error in the Hamming code
   signal Update_Hamming : std_logic;     -- Indicates Hamming has been corrected

   signal Data_corrupt0 : std_logic;       -- Indicates data/Haming uncorrectable
   signal Data_corrupt1 : std_logic;       -- Indicates data/Haming uncorrectable
   signal Data_corrupt2 : std_logic;       -- Indicates data/Haming uncorrectable
   signal Uncorrectable_error : std_logic;

   signal nWR_in : std_logic;
   signal nWR_out : std_logic;
   signal nWR_out_delay : std_logic;
   signal nWR_out_inv : std_logic;

   signal nRD_out : std_logic;
   signal nRD_in : std_logic;
   signal nRD_in_inv : std_logic;
   signal nRD_in_delay : std_logic;

------------------------ REGISTERS FOR SETTING UP EDAC -----------------

   signal PM_start_reg : std_logic_vector ( 15 downto 0 );
   signal PM_end_reg : std_logic_vector ( 15 downto 0 );
   signal Hamming_start_reg : std_logic_vector ( 15 downto 0 );

   signal EDAC_setup_reg : std_logic_vector ( 15 downto 0 );

   signal Single_errors_reg : std_logic_vector ( 15 downto 0 );
   signal Double_errors_reg : std_logic_vector ( 15 downto 0 );

   signal Register_CS : std_logic;
   signal Register_out : std_logic_vector ( 15 downto 0 );
   signal Register_in : std_logic_vector ( 15 downto 0 );

begin

--=========================================================

   state_process : process( clk, Reset, nRD_out )

      variable Main_state : Main_states;
      variable Read_state : Read_states;
      variable Generate_ham_state : Generate_ham_states;
      variable Write_state : Write_states;
      variable Word_count : Integer range 0 to 2;    -- Counts the words read/written
      variable Data_type : Data_types;
      variable Delay : Integer range 0 to 2;   -- Counter to generate delay

   begin

      if Reset = '1' then               -- Assign startup values

         Read_state := Request_HB;
         Write_state := Request_HB;
         Word_count := 0;
         Data_type := Data;
         Delay := 0;

         Addr_count <= ( others => '0' );
```

```
        PM_read <= ( others => '0' );
        Hamming_read <= ( others => '0' );

        PM_start_reg <= ( others => '0' );       -- Start values for registers
        PM_end_reg <= ( others => '1' );
        Hamming_start_reg <= ( others => '1' );
        EDAC_setup_reg <= ( others => '0' );
        Single_errors_reg <= ( others => '0' );
        Double_errors_reg <= ( others => '0' );

        nHBR <= '1';
        nCS <= '1';
        nSBTS <= '1';

        nRD_out <= '1';
        nWR_out <= '1';
        nSW <= '1';
        PAGE <= '0';

        nMS <= ( others => '1' );
        ADRCLK <= '0';
        nDMAG1 <= '1';
        nDMAG2 <= '1';

        Main_state := Wait_start;

     elsif clk'event and clk = '1' then

        case Main_state is
-------------------------------------------------------------

        when Wait_start =>            -- Wait until EDAC enable is set

           if EDAC_setup_reg( 0 ) = '1' then       -- EDAC enable = '1' ?
              Main_state := Read_PM;
              Data_type := Data;
              Addr_out <= Next_PM_addr;
           end if;
-------------------------------------------------------------

        when Read_PM =>        -- State to read data from DSP

           case Read_state is
           when Request_HB =>      -- Do host bus request
              nHBR <= '0';
              nCS <= '0';
              Read_state := Wait_HBG;


           when Wait_HBG =>        -- Wait for host bus grant and enable outputs
              if nHBG = '0' then
                 nRD_out <= '0';
                 Read_state := Wait_REDY;
              end if;


           when Wait_REDY =>        -- Wait for REDY and latch data
              if REDY ='1' then
                 nRD_out <= '1';
                 Read_state := Store_words;
              end if;


           when Store_words =>      -- Stores words in 48-bit buffer

              if Data_type = Data then

                 case Word_count is
                 when 0 =>
                    PM_read ( 47 downto 32 ) <= Data_in;
                    Word_count := 1;
```

```
                Read_state := Wait_HBG;

            when 1 =>
                PM_read ( 31 downto 16 ) <= Data_in;
                Word_count := 2;
                Read_state := Wait_HBG;

            when 2 =>
                PM_read ( 15 downto 0 ) <= Data_in;
                Word_count := 0;
                Read_state := Release_HB;

            end case;

        else

            case Word_count is
            when 0 =>
                Hamming_read ( 47 downto 32 ) <= Data_in;
                Word_count := 1;
                Read_state := Wait_HBG;

            when 1 =>
                Hamming_read ( 31 downto 16 ) <= Data_in;
                Word_count := 2;
                Read_state := Wait_HBG;

            when 2 =>
                Hamming_read ( 15 downto 0 ) <= Data_in;
                Word_count := 0;
                Read_state := Release_HB;

            end case;

        end if;


    when Release_HB =>        -- Release the host bus + calculate next addr

        nHBR <= '1';
        nCS <= '1';

        Read_state := Request_HB;

        if Data_type = Data then
            Addr_out <= Next_Hamming_addr;
            Data_type := Hamming;
            Main_state := Read_PM;
        else
            Main_state := Generate_ham;
        end if;

    end case;

------------------------------------------------------------


    when Generate_ham =>    -- State to generate/check data with Hamming code

        case Generate_ham_state is

        when Calculate_ham =>    -- Wait for asynchronous logic to settle

            Generate_ham_state := Check_data;

--------------- Write PM_write to DSP if corrected -----------------


            when Check_data =>

                if Uncorrectable_error = '1' then    -- Signal data corrupt, skip this location
```

```
                Double_errors_reg <= Double_errors_reg + 1;  -- Increment error count

                if Addr_dif = Addr_count then            -- Test for end of PM
                   Addr_count <= ( others => '0' );
                else Addr_count <= Addr_count + 1;       -- Increment address counter
                end if;

                Generate_ham_state := Calculate_ham;
                Main_state := Wait_start;

             else

                if Update_data = '1' then   -- Write corrected data to PM

                   Single_errors_reg <= Single_errors_reg + Data_error0 +
                                        Data_error1 + Data_error2;

                   Data_type := Data;
                   Main_state := Write_PM;

                end if;

                Generate_ham_state := Check_Hamming;

             end if;


----------------- Write Hamming_write to DSP if corrected ------------------

          when Check_Hamming =>

             if Update_Hamming = '1' then

                Single_errors_reg <= Single_errors_reg + Hamming_error0 +
                                     Hamming_error1 + Hamming_error2;

                Data_type := Hamming;
                Main_state := Write_PM;

             else

                if Addr_dif = Addr_count then           -- Test for end of PM
                   Addr_count <= ( others => '0' );
                else Addr_count <= Addr_count +1;       -- Increment address counter
                end if;

                Main_state := Wait_start;

             end if;

             Generate_ham_state := Calculate_ham;

          end case;


---------------- Write 48 bits in three 16-bit cycles ----------------------

       when Write_PM =>

          case Write_state is
          when Request_HB =>          -- Request the host bus

             nHBR <= '0';
             nCS <= '0';
             Write_state := Write_words;

          when Write_words =>         -- Determine next word to write

             if Data_type = Data then

                case Word_count is
                when 0 =>
                   Data_out <= PM_write ( 47 downto 32 );
                   Word_count := 1;
```

```
                when 1 =>
                   Data_out <= PM_write ( 31 downto 16 );
                   Word_count := 2;

                when 2 =>
                   Data_out <= PM_write ( 15 downto 0 );
                   Word_count := 0;

                end case;

             else

                case Word_count is
                when 0 =>
                   Data_out <= Hamming_write ( 47 downto 32 );
                   Word_count := 1;

                when 1 =>
                   Data_out <= Hamming_write ( 31 downto 16 );
                   Word_count := 2;

                when 2 =>
                   Data_out <= Hamming_write ( 15 downto 0 );
                   Word_count := 0;

                end case;

             end if;

             Write_state := Wait_HBG;

          when Wait_HBG =>          -- Wait for the host bus grant signal

             if nHBG = '0' then
                nWR_out <= '0';
                Write_state := Wait_REDY;
             end if;

          when Wait_REDY =>         -- Wait for REDY signal

             if REDY = '1' then
                nWR_out <= '1';
                if Word_count = 0 then Write_state := Release_HB;
                                  else Write_state := Write_words;
                end if;
             end if;

          when Release_HB =>        -- Release the host bus

             nHBR <= '1';
             nCS <= '1';
             Write_state := Request_HB;

             if Data_type = Data then
                Main_state := Generate_ham;
             else
                if Addr_dif = Addr_count then           -- Test for end of PM
                   Addr_count <= ( others => '0' );
                else Addr_count <= Addr_count +1;       -- Increment address counter
                end if;

                Main_state := Wait_start;
             end if;

          end case;

       end case;

------------ Write new values to registers ------------------------------

       if ( nWR_in = '0' and Register_CS = '1' and Reset = '0' ) then
```

```
            case conv_integer( Addr_port( 2 downto 0 ) ) is
            when 16#00# =>
               PM_start_reg <= Data_port( 15 downto 0 );
            when 16#01# =>
               PM_end_reg <= Data_port( 15 downto 0 );
            when 16#02# =>
               Hamming_start_reg <= Data_port( 15 downto 0 );
            when 16#03# =>
               EDAC_setup_reg <= Data_port( 15 downto 0 );
            when others =>
            end case;

         end if;

      end if;

   end process;

--=================== Process to read in data ============================

   Read_process : process( nRD_out )
    begin

      if ( nRD_out'event and nRD_out='1' ) then Data_in <= Data_port;
       end if;

   end process;

--=======================================================

----- Assign values to address and data port according to the state ---------
----- of the nWR, nRD and nHBG pins                                 ---------

   -- The address must be valid for 2ns, and data for 1ns after nWR goes high
   -- Method to get a > 2ns delay after nWR goes high

   nWR_out_inv <= not nWR_out;
   nWR_out_delay <= not NWR_out_inv;

   nRD_in_inv <= not nRD_in;
   nRD_in_delay <= not nRD_in_inv;

   nWR_port <= nWR_out when ( nHBG='0' and Reset = '0' ) else
               'Z';
   nRD_port <= nRD_out when ( nHBG='0' and Reset = '0' ) else
               'Z';

   nRD_in <= nRD_port when ( nHBG = '1' and Reset = '0' ) else
             '1';
   nWR_in <= nWR_port when ( nHBG = '1' and Reset = '0' ) else
             '1';

   Data_port <= Data_out when ( nWR_out_delay = '0' and nHBG = '0' and Reset = '0' ) else
                Register_out when ( nRD_in_delay ='0' and Register_CS = '1' and
                                    Reset = '0' ) else
                ( others => 'Z' );

   Addr_port <= Addr_out when nHBG='0' and Reset = '0' and ( nRD_out='0' or
                              nWR_out_delay='0' ) else
                ( others => 'Z' );

   Addr_in <= Addr_port when ( nHBG = '1' and Reset = '0' ) else
              ( others => '0' );

   Addr_dif <= PM_end_reg - PM_start_reg;          -- Used to check if all memory checked

   Next_PM_addr <= PM_start_reg + Addr_count;

   Next_Hamming_addr <= Hamming_start_reg + Addr_count( 15 downto 1 );  -- shr( Addr_count, "1" );


-------- Address decoder for reading / writing the registers --------------
```

```
   with conv_integer( Addr_port( 23 downto 0 ) ) select
   Register_CS <= '1' when 16#400000#,
                  '1' when 16#400001#,
                  '1' when 16#400002#,
                  '1' when 16#400003#,
                  '1' when 16#400004#,
                  '1' when 16#400005#,
                  '0' when others;


-------- Writes register to temporary buffer, according to address --------


   with conv_integer( Addr_port( 23 downto 0 ) ) select
   Register_out <= PM_start_reg      when 16#400000#,
                   PM_end_reg        when 16#400001#,
                   Hamming_start_reg when 16#400002#,
                   EDAC_setup_reg    when 16#400003#,
                   Single_errors_reg when 16#400004#,
                   Double_errors_reg when 16#400005#,
                   ( others => '0' ) when others;


------ Calculate new Hamming codes from data and generate syndromes --------


   Ham_new0(0) <= ( PM_read(0) xor PM_read(1) ) xor ( PM_read(3) xor PM_read(4) )
                  xor ( PM_read(8) xor PM_read(9) ) xor ( PM_read(10) xor PM_read(13) );
   Ham_new0(1) <= ( PM_read(0) xor PM_read(2) ) xor ( PM_read(3) xor PM_read(5) )
                  xor ( PM_read(6) xor PM_read(8) ) xor ( PM_read(11) xor PM_read(14) );
   Ham_new0(2) <= ( PM_read(1) xor PM_read(2) ) xor ( PM_read(4) xor PM_read(5) )
                  xor ( PM_read(7) xor PM_read(9) ) xor ( PM_read(12) xor PM_read(15) );
   Ham_new0(3) <= ( PM_read(0) xor PM_read(1) ) xor ( PM_read(2) xor PM_read(6) )
                  xor ( PM_read(7) xor PM_read(10) ) xor ( PM_read(11) xor PM_read(12) );
   Ham_new0(4) <= ( PM_read(3) xor PM_read(4) ) xor ( PM_read(5) xor PM_read(6) )
                  xor ( PM_read(7) xor PM_read(13) ) xor ( PM_read(14) xor PM_read(15) );
   Ham_new0(5) <= ( PM_read(8) xor PM_read(9) ) xor ( PM_read(10) xor PM_read(11) )
                  xor ( PM_read(12) xor PM_read(13) ) xor ( PM_read(14) xor PM_read(15) );

                                                      -- Generate the syndrome
   Ham_syndrome0 <= Ham_new0 xor Hamming_read( 5 downto 0 ) when Addr_count( 0 ) = '0' else
                    Ham_new0 xor Hamming_read( 29 downto 24 );

   Ham_new1(0) <= ( PM_read(0+16) xor PM_read(1+16) ) xor ( PM_read(3+16) xor PM_read(4+16) )
                  xor ( PM_read(8+16) xor PM_read(9+16) ) xor ( PM_read(10+16) xor PM_read(13+16) );
   Ham_new1(1) <= ( PM_read(0+16) xor PM_read(2+16) ) xor ( PM_read(3+16) xor PM_read(5+16) )
                  xor ( PM_read(6+16) xor PM_read(8+16) ) xor ( PM_read(11+16) xor PM_read(14+16) );
   Ham_new1(2) <= ( PM_read(1+16) xor PM_read(2+16) ) xor ( PM_read(4+16) xor PM_read(5+16) )
                  xor ( PM_read(7+16) xor PM_read(9+16) ) xor ( PM_read(12+16) xor PM_read(15+16) );
   Ham_new1(3) <= ( PM_read(0+16) xor PM_read(1+16) ) xor ( PM_read(2+16) xor PM_read(6+16) )
                  xor ( PM_read(7+16) xor PM_read(10+16) ) xor ( PM_read(11+16) xor PM_read(12+16) );
   Ham_new1(4) <= ( PM_read(3+16) xor PM_read(4+16) ) xor ( PM_read(5+16) xor PM_read(6+16) )
                  xor ( PM_read(7+16) xor PM_read(13+16) ) xor ( PM_read(14+16) xor PM_read(15+16) );
   Ham_new1(5) <= ( PM_read(8+16) xor PM_read(9+16) ) xor ( PM_read(10+16) xor PM_read(11+16) )
                  xor ( PM_read(12+16) xor PM_read(13+16) ) xor ( PM_read(14+16) xor PM_read(15+16) );

                      -- Generate the syndrome
   Ham_syndrome1 <= Ham_new1 xor Hamming_read( 11 downto 6 ) when Addr_count( 0 ) = '0' else
                    Ham_new1 xor Hamming_read( 35 downto 30 );


   Ham_new2(0) <= ( PM_read(0+32) xor PM_read(1+32) ) xor ( PM_read(3+32) xor PM_read(4+32) )
                  xor ( PM_read(8+32) xor PM_read(9+32) ) xor ( PM_read(10+32) xor PM_read(13+32) );
   Ham_new2(1) <= ( PM_read(0+32) xor PM_read(2+32) ) xor ( PM_read(3+32) xor PM_read(5+32) )
                  xor ( PM_read(6+32) xor PM_read(8+32) ) xor ( PM_read(11+32) xor PM_read(14+32) );
   Ham_new2(2) <= ( PM_read(1+32) xor PM_read(2+32) ) xor ( PM_read(4+32) xor PM_read(5+32) )
                  xor ( PM_read(7+32) xor PM_read(9+32) ) xor ( PM_read(12+32) xor PM_read(15+32) );
   Ham_new2(3) <= ( PM_read(0+32) xor PM_read(1+32) ) xor ( PM_read(2+32) xor PM_read(6+32) )
                  xor ( PM_read(7+32) xor PM_read(10+32) ) xor ( PM_read(11+32) xor PM_read(12+32) );
   Ham_new2(4) <= ( PM_read(3+32) xor PM_read(4+32) ) xor ( PM_read(5+32) xor PM_read(6+32) )
                  xor ( PM_read(7+32) xor PM_read(13+32) ) xor ( PM_read(14+32) xor PM_read(15+32) );
   Ham_new2(5) <= ( PM_read(8+32) xor PM_read(9+32) ) xor ( PM_read(10+32) xor PM_read(11+32) )
                  xor ( PM_read(12+32) xor PM_read(13+32) ) xor ( PM_read(14+32) xor PM_read(15+32) );
```

```
                    -- Generate the syndrome
   Ham_syndrome2 <= Ham_new2 xor Hamming_read( 17 downto 12 ) when Addr_count( 0 ) = '0' else
                    Ham_new2 xor Hamming_read( 41 downto 36 );


------------------- Determine if errors / type of errors -------------------


   with Ham_syndrome0 select
   Hamming_error0 <= '0' when "000000",          -- No errors
                     '1' when "000001",          -- Error in Hamming bit 0
                     '1' when "000010",          -- Error in Hamming bit 1
                     '1' when "000100",          -- Error in Hamming bit 2
                     '1' when "001000",          -- Error in Hamming bit 3
                     '1' when "010000",          -- Error in Hamming bit 4
                     '1' when "100000",          -- Error in Hamming bit 5
                     '0' when others;

   with Ham_syndrome0 select
   Data_error0 <= '0' when "000000",          -- No errors
                  '1' when "001011",          -- Error in data bit 0
                  '1' when "001101",          -- Error in data bit 1
                  '1' when "001110",          -- Error in data bit 2
                  '1' when "010011",          -- Error in data bit 3
                  '1' when "010101",          -- Error in data bit 4
                  '1' when "010110",          -- Error in data bit 5
                  '1' when "011010",          -- Error in data bit 6
                  '1' when "011100",          -- Error in data bit 7
                  '1' when "100011",          -- Error in data bit 8
                  '1' when "100101",          -- Error in data bit 9
                  '1' when "101001",          -- Error in data bit 10
                  '1' when "101010",          -- Error in data bit 11
                  '1' when "101100",          -- Error in data bit 12
                  '1' when "110001",          -- Error in data bit 13
                  '1' when "110010",          -- Error in data bit 14
                  '1' when "110100",          -- Error in data bit 15
                  '0' when others;

   with Ham_syndrome0 select
   Data_corrupt0 <= '0' when "000000",          -- No errors
                    '0' when "000001",          -- Error in Hamming bit 0
                    '0' when "000010",          -- Error in Hamming bit 1
                    '0' when "000100",          -- Error in Hamming bit 2
                    '0' when "001000",          -- Error in Hamming bit 3
                    '0' when "010000",          -- Error in Hamming bit 4
                    '0' when "100000",          -- Error in Hamming bit 5
                    '0' when "001011",          -- Error in data bit 0
                    '0' when "001101",          -- Error in data bit 1
                    '0' when "001110",          -- Error in data bit 2
                    '0' when "010011",          -- Error in data bit 3
                    '0' when "010101",          -- Error in data bit 4
                    '0' when "010110",          -- Error in data bit 5
                    '0' when "011010",          -- Error in data bit 6
                    '0' when "011100",          -- Error in data bit 7
                    '0' when "100011",          -- Error in data bit 8
                    '0' when "100101",          -- Error in data bit 9
                    '0' when "101001",          -- Error in data bit 10
                    '0' when "101010",          -- Error in data bit 11
                    '0' when "101100",          -- Error in data bit 12
                    '0' when "110001",          -- Error in data bit 13
                    '0' when "110010",          -- Error in data bit 14
                    '0' when "110100",          -- Error in data bit 15
                    '1' when others;


   with Ham_syndrome1 select
   Hamming_error1 <= '0' when "000000",          -- No errors
                     '1' when "000001",          -- Error in Hamming bit 0
                     '1' when "000010",          -- Error in Hamming bit 1
                     '1' when "000100",          -- Error in Hamming bit 2
                     '1' when "001000",          -- Error in Hamming bit 3
                     '1' when "010000",          -- Error in Hamming bit 4
                     '1' when "100000",          -- Error in Hamming bit 5
```

```
                '0' when others;

with Ham_syndrome1 select
Data_error1 <= '0' when "000000",          -- No errors
               '1' when "001011",          -- Error in data bit 0
               '1' when "001101",          -- Error in data bit 1
               '1' when "001110",          -- Error in data bit 2
               '1' when "010011",          -- Error in data bit 3
               '1' when "010101",          -- Error in data bit 4
               '1' when "010110",          -- Error in data bit 5
               '1' when "011010",          -- Error in data bit 6
               '1' when "011100",          -- Error in data bit 7
               '1' when "100011",          -- Error in data bit 8
               '1' when "100101",          -- Error in data bit 9
               '1' when "101001",          -- Error in data bit 10
               '1' when "101010",          -- Error in data bit 11
               '1' when "101100",          -- Error in data bit 12
               '1' when "110001",          -- Error in data bit 13
               '1' when "110010",          -- Error in data bit 14
               '1' when "110100",          -- Error in data bit 15
               '0' when others;

with Ham_syndrome1 select
Data_corrupt1 <= '0' when "000000",          -- No errors
                 '0' when "000001",          -- Error in Hamming bit 0
                 '0' when "000010",          -- Error in Hamming bit 1
                 '0' when "000100",          -- Error in Hamming bit 2
                 '0' when "001000",          -- Error in Hamming bit 3
                 '0' when "010000",          -- Error in Hamming bit 4
                 '0' when "100000",          -- Error in Hamming bit 5
                 '0' when "001011",          -- Error in data bit 0
                 '0' when "001101",          -- Error in data bit 1
                 '0' when "001110",          -- Error in data bit 2
                 '0' when "010011",          -- Error in data bit 3
                 '0' when "010101",          -- Error in data bit 4
                 '0' when "010110",          -- Error in data bit 5
                 '0' when "011010",          -- Error in data bit 6
                 '0' when "011100",          -- Error in data bit 7
                 '0' when "100011",          -- Error in data bit 8
                 '0' when "100101",          -- Error in data bit 9
                 '0' when "101001",          -- Error in data bit 10
                 '0' when "101010",          -- Error in data bit 11
                 '0' when "101100",          -- Error in data bit 12
                 '0' when "110001",          -- Error in data bit 13
                 '0' when "110010",          -- Error in data bit 14
                 '0' when "110100",          -- Error in data bit 15
                 '1' when others;


with Ham_syndrome2 select
Hamming_error2 <= '0' when "000000",          -- No errors
                  '1' when "000001",          -- Error in Hamming bit 0
                  '1' when "000010",          -- Error in Hamming bit 1
                  '1' when "000100",          -- Error in Hamming bit 2
                  '1' when "001000",          -- Error in Hamming bit 3
                  '1' when "010000",          -- Error in Hamming bit 4
                  '1' when "100000",          -- Error in Hamming bit 5
                  '0' when others;

with Ham_syndrome2 select
Data_error2 <= '0' when "000000",          -- No errors
               '1' when "001011",          -- Error in data bit 0
               '1' when "001101",          -- Error in data bit 1
               '1' when "001110",          -- Error in data bit 2
               '1' when "010011",          -- Error in data bit 3
               '1' when "010101",          -- Error in data bit 4
               '1' when "010110",          -- Error in data bit 5
               '1' when "011010",          -- Error in data bit 6
               '1' when "011100",          -- Error in data bit 7
               '1' when "100011",          -- Error in data bit 8
               '1' when "100101",          -- Error in data bit 9
               '1' when "101001",          -- Error in data bit 10
               '1' when "101010",          -- Error in data bit 11
               '1' when "101100",          -- Error in data bit 12
```

```
                '1' when "110001",          -- Error in data bit 13
                '1' when "110010",          -- Error in data bit 14
                '1' when "110100",          -- Error in data bit 15
                '0' when others;

   with Ham_syndrome2 select
   Data_corrupt2 <= '0' when "000000",          -- No errors
                    '0' when "000001",          -- Error in Hamming bit 0
                    '0' when "000010",          -- Error in Hamming bit 1
                    '0' when "000100",          -- Error in Hamming bit 2
                    '0' when "001000",          -- Error in Hamming bit 3
                    '0' when "010000",          -- Error in Hamming bit 4
                    '0' when "100000",          -- Error in Hamming bit 5
                    '0' when "001011",          -- Error in data bit 0
                    '0' when "001101",          -- Error in data bit 1
                    '0' when "001110",          -- Error in data bit 2
                    '0' when "010011",          -- Error in data bit 3
                    '0' when "010101",          -- Error in data bit 4
                    '0' when "010110",          -- Error in data bit 5
                    '0' when "011010",          -- Error in data bit 6
                    '0' when "011100",          -- Error in data bit 7
                    '0' when "100011",          -- Error in data bit 8
                    '0' when "100101",          -- Error in data bit 9
                    '0' when "101001",          -- Error in data bit 10
                    '0' when "101010",          -- Error in data bit 11
                    '0' when "101100",          -- Error in data bit 12
                    '0' when "110001",          -- Error in data bit 13
                    '0' when "110010",          -- Error in data bit 14
                    '0' when "110100",          -- Error in data bit 15
                    '1' when others;


   Update_data <= ( Data_error0 or Data_error1 or Data_error2 ) and
                  not Uncorrectable_error;

   Update_Hamming <= ( Hamming_error0 or Hamming_error1 or Hamming_error2 ) and
                     not Uncorrectable_error;

   Uncorrectable_error <= Data_corrupt0 or Data_corrupt1 or Data_corrupt2;


-------------- Write corrected data / Hamming to temporary buffer ----------


   with  Ham_syndrome0 select
   PM_write( 15 downto 0 ) <=
      PM_read( 15 downto 0 ) xor "0000000000000001" when "001011",  -- Error in data bit 0
      PM_read( 15 downto 0 ) xor "0000000000000010" when "001101",  -- Error in data bit 1
      PM_read( 15 downto 0 ) xor "0000000000000100" when "001110",  -- Error in data bit 2
      PM_read( 15 downto 0 ) xor "0000000000001000" when "010011",  -- Error in data bit 3
      PM_read( 15 downto 0 ) xor "0000000000010000" when "010101",  -- Error in data bit 4
      PM_read( 15 downto 0 ) xor "0000000000100000" when "010110",  -- Error in data bit 5
      PM_read( 15 downto 0 ) xor "0000000001000000" when "011010",  -- Error in data bit 6
      PM_read( 15 downto 0 ) xor "0000000010000000" when "011100",  -- Error in data bit 7
      PM_read( 15 downto 0 ) xor "0000000100000000" when "100011",  -- Error in data bit 8
      PM_read( 15 downto 0 ) xor "0000001000000000" when "100101",  -- Error in data bit 9
      PM_read( 15 downto 0 ) xor "0000010000000000" when "101001",  -- Error in data bit 10
      PM_read( 15 downto 0 ) xor "0000100000000000" when "101010",  -- Error in data bit 11
      PM_read( 15 downto 0 ) xor "0001000000000000" when "101100",  -- Error in data bit 12
      PM_read( 15 downto 0 ) xor "0010000000000000" when "110001",  -- Error in data bit 13
      PM_read( 15 downto 0 ) xor "0100000000000000" when "110010",  -- Error in data bit 14
      PM_read( 15 downto 0 ) xor "1000000000000000" when "110100",  -- Error in data bit 15
      PM_read( 15 downto 0 ) when others;

   with  Ham_syndrome1 select
   PM_write( 31 downto 16 ) <=
      PM_read( 31 downto 16 ) xor "0000000000000001" when "001011",  -- Error in data bit 0
      PM_read( 31 downto 16 ) xor "0000000000000010" when "001101",  -- Error in data bit 1
      PM_read( 31 downto 16 ) xor "0000000000000100" when "001110",  -- Error in data bit 2
      PM_read( 31 downto 16 ) xor "0000000000001000" when "010011",  -- Error in data bit 3
      PM_read( 31 downto 16 ) xor "0000000000010000" when "010101",  -- Error in data bit 4
      PM_read( 31 downto 16 ) xor "0000000000100000" when "010110",  -- Error in data bit 5
      PM_read( 31 downto 16 ) xor "0000000001000000" when "011010",  -- Error in data bit 6
      PM_read( 31 downto 16 ) xor "0000000010000000" when "011100",  -- Error in data bit 7
```

```
   PM_read( 31 downto 16 ) xor "0000000100000000" when "100011",  -- Error in data bit 8
   PM_read( 31 downto 16 ) xor "0000001000000000" when "100101",  -- Error in data bit 9
   PM_read( 31 downto 16 ) xor "0000010000000000" when "101001",  -- Error in data bit 10
   PM_read( 31 downto 16 ) xor "0000100000000000" when "101010",  -- Error in data bit 11
   PM_read( 31 downto 16 ) xor "0001000000000000" when "101100",  -- Error in data bit 12
   PM_read( 31 downto 16 ) xor "0010000000000000" when "110001",  -- Error in data bit 13
   PM_read( 31 downto 16 ) xor "0100000000000000" when "110010",  -- Error in data bit 14
   PM_read( 31 downto 16 ) xor "1000000000000000" when "110100",  -- Error in data bit 15
   PM_read( 31 downto 16 ) when others;

with  Ham_syndrome2 select
PM_write( 47 downto 32 ) <=
   PM_read( 47 downto 32 ) xor "0000000000000001" when "001011",  -- Error in data bit 0
   PM_read( 47 downto 32 ) xor "0000000000000010" when "001101",  -- Error in data bit 1
   PM_read( 47 downto 32 ) xor "0000000000000100" when "001110",  -- Error in data bit 2
   PM_read( 47 downto 32 ) xor "0000000000001000" when "010011",  -- Error in data bit 3
   PM_read( 47 downto 32 ) xor "0000000000010000" when "010101",  -- Error in data bit 4
   PM_read( 47 downto 32 ) xor "0000000000100000" when "010110",  -- Error in data bit 5
   PM_read( 47 downto 32 ) xor "0000000001000000" when "011010",  -- Error in data bit 6
   PM_read( 47 downto 32 ) xor "0000000010000000" when "011100",  -- Error in data bit 7
   PM_read( 47 downto 32 ) xor "0000000100000000" when "100011",  -- Error in data bit 8
   PM_read( 47 downto 32 ) xor "0000001000000000" when "100101",  -- Error in data bit 9
   PM_read( 47 downto 32 ) xor "0000010000000000" when "101001",  -- Error in data bit 10
   PM_read( 47 downto 32 ) xor "0000100000000000" when "101010",  -- Error in data bit 11
   PM_read( 47 downto 32 ) xor "0001000000000000" when "101100",  -- Error in data bit 12
   PM_read( 47 downto 32 ) xor "0010000000000000" when "110001",  -- Error in data bit 13
   PM_read( 47 downto 32 ) xor "0100000000000000" when "110010",  -- Error in data bit 14
   PM_read( 47 downto 32 ) xor "1000000000000000" when "110100",  -- Error in data bit 15
   PM_read( 47 downto 32 ) when others;


Ham_decode0 <= Addr_count(0) & Ham_syndrome0;       -- Combination of signals
Ham_decode1 <= Addr_count(0) & Ham_syndrome1;       -- to make decoding easier
Ham_decode2 <= Addr_count(0) & Ham_syndrome2;

with Ham_decode0 select
Hamming_write( 5 downto 0 ) <=
   Hamming_read( 5 downto 0 ) xor "000001" when "0000001",     -- Error in Hamming bit 0
   Hamming_read( 5 downto 0 ) xor "000010" when "0000010",     -- Error in Hamming bit 1
   Hamming_read( 5 downto 0 ) xor "000100" when "0000100",     -- Error in Hamming bit 2
   Hamming_read( 5 downto 0 ) xor "001000" when "0001000",     -- Error in Hamming bit 3
   Hamming_read( 5 downto 0 ) xor "010000" when "0010000",     -- Error in Hamming bit 4
   Hamming_read( 5 downto 0 ) xor "100000" when "0100000",     -- Error in Hamming bit 5
   Hamming_read( 5 downto 0 ) when others;

with Ham_decode0 select
Hamming_write( 29 downto 24 ) <=
   Hamming_read( 29 downto 24 ) xor "000001" when "1000001",     -- Error in Hamming bit 0
   Hamming_read( 29 downto 24 ) xor "000010" when "1000010",     -- Error in Hamming bit 1
   Hamming_read( 29 downto 24 ) xor "000100" when "1000100",     -- Error in Hamming bit 2
   Hamming_read( 29 downto 24 ) xor "001000" when "1001000",     -- Error in Hamming bit 3
   Hamming_read( 29 downto 24 ) xor "010000" when "1010000",     -- Error in Hamming bit 4
   Hamming_read( 29 downto 24 ) xor "100000" when "1100000",     -- Error in Hamming bit 5
   Hamming_read( 29 downto 24 ) when others;

with Ham_decode1 select
Hamming_write( 11 downto 6 ) <=
   Hamming_read( 11 downto 6 ) xor "000001" when "0000001",     -- Error in Hamming bit 0
   Hamming_read( 11 downto 6 ) xor "000010" when "0000010",     -- Error in Hamming bit 1
   Hamming_read( 11 downto 6 ) xor "000100" when "0000100",     -- Error in Hamming bit 2
   Hamming_read( 11 downto 6 ) xor "001000" when "0001000",     -- Error in Hamming bit 3
   Hamming_read( 11 downto 6 ) xor "010000" when "0010000",     -- Error in Hamming bit 4
   Hamming_read( 11 downto 6 ) xor "100000" when "0100000",     -- Error in Hamming bit 5
   Hamming_read( 11 downto 6 ) when others;

with Ham_decode1 select
Hamming_write( 35 downto 30 ) <=
   Hamming_read( 35 downto 30 ) xor "000001" when "1000001",     -- Error in Hamming bit 0
   Hamming_read( 35 downto 30 ) xor "000010" when "1000010",     -- Error in Hamming bit 1
   Hamming_read( 35 downto 30 ) xor "000100" when "1000100",     -- Error in Hamming bit 2
   Hamming_read( 35 downto 30 ) xor "001000" when "1001000",     -- Error in Hamming bit 3
   Hamming_read( 35 downto 30 ) xor "010000" when "1010000",     -- Error in Hamming bit 4
   Hamming_read( 35 downto 30 ) xor "100000" when "1100000",     -- Error in Hamming bit 5
   Hamming_read( 35 downto 30 ) when others;
```

```
   with Ham_decode2 select
   Hamming_write( 17 downto 12 ) <=
      Hamming_read( 17 downto 12 ) xor "000001" when "0000001",     -- Error in Hamming bit 0
      Hamming_read( 17 downto 12 ) xor "000010" when "0000010",     -- Error in Hamming bit 1
      Hamming_read( 17 downto 12 ) xor "000100" when "0000100",     -- Error in Hamming bit 2
      Hamming_read( 17 downto 12 ) xor "001000" when "0001000",     -- Error in Hamming bit 3
      Hamming_read( 17 downto 12 ) xor "010000" when "0010000",     -- Error in Hamming bit 4
      Hamming_read( 17 downto 12 ) xor "100000" when "0100000",     -- Error in Hamming bit 5
      Hamming_read( 17 downto 12 ) when others;

   with Ham_decode2 select
   Hamming_write( 41 downto 36 ) <=
      Hamming_read( 41 downto 36 ) xor "000001" when "1000001",     -- Error in Hamming bit 0
      Hamming_read( 41 downto 36 ) xor "000010" when "1000010",     -- Error in Hamming bit 1
      Hamming_read( 41 downto 36 ) xor "000100" when "1000100",     -- Error in Hamming bit 2
      Hamming_read( 41 downto 36 ) xor "001000" when "1001000",     -- Error in Hamming bit 3
      Hamming_read( 41 downto 36 ) xor "010000" when "1010000",     -- Error in Hamming bit 4
      Hamming_read( 41 downto 36 ) xor "100000" when "1100000",     -- Error in Hamming bit 5
      Hamming_read( 41 downto 36 ) when others;

   Hamming_write( 23 downto 18 ) <= "000000";          -- Not used for Hamming codes
   Hamming_write( 47 downto 42 ) <= "000000";

-------------------------------------------------------------

end behavioral;
```